RADC-TR-88-137
Final Technical Report
July 1988

ARCHITECTURAL STUDY OF ADAPTIVE ALGORITHMS FOR ADAPTIVE BEAM COMMUNICATION ANTENNAS

Space Tech Corporation

Michael Andrews and Robert Edward Boring

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC TR-88-137 has been reviewed and is approved for publication.

APPROVED: Robert S. Kr.

ROBERT A. SHORE
Project Engineer

APPROVED: John K. Schindler

JOHN K. SCHINDLER
Acting Director of Electromagnetics

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (EEAS) Hanscom AFB MA 01731-5000. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific doucment require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) FR ITR-SBIR-85-1-R	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-137
6a. NAME OF PERFORMING ORGANIZATION Space Tech Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (EEAS)
6c. ADDRESS (City, State, and ZIP Code) 2324 Manchester Court Ft. Collins CO 80526	7b. ADDRESS (City, State, and ZIP Code) Hanscom AFB MA 01731-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) EEAS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-85-C-0144
8c. ADDRESS (City, State, and ZIP Code) Hanscom AFB MA 01731-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 65502F; PROJECT NO. 3005; TASK NO RA; WORK UNIT ACCESSION NO 62

11. TITLE (Include Security Classification)
ARCHITECTURAL STUDY OF ADAPTIVE ALGORITHMS FOR ADAPTIVE BEAM COMMUNICATION ANTENNAS

12. PERSONAL AUTHOR(S)
Michael Andrews, Robert Edward Boring

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Jul 85 TO Jan 86	July 1988	140

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	01, 03	
20	14	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Adaptive Beamforming; Least Squares Minimization
Systolic Array; Givens Rotations, (jrd)
Redundant Numbers

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A comparative architecture study was performed in order to implement the scaled Given rotation solution to the least-squares minimization problem. Three architectures are examined: (a) Conventional Systolic Array, (b) SBNR Systolic Array, and (c) Distributed Arithmetic Systolic Array.

Important considerations in adaptive beamforming algorithm to architecture mapping are also considered. The issues addressed included oversized array processing on fixed sized systolic arrays, error analysis, and design tools. Keywords:

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Robert A. Shore	22b. TELEPHONE (Include Area Code) (617) 377-2058 22c. OFFICE SYMBOL RADC (EEAS)

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Table of Contents

1.0 Abstract

A comparative architecture study was performed in order to implement the scaled Givens rotation solution to the least-squares minimization problem. Three architectures are examined:

a. Conventional Systolic Array
b. SBNR Systolic Array
c. Distributed Arithmetic Systolic Array

Important considerations in adaptive beamforming algorithm to architecture mapping are also considered. The issues addressed include oversized array processing on fixed sized systolic arrays, error analysis, and design tools. A discussion of Ada as a grammar for specification of adaptive beamform algorithms for architecture design, simulation, and implementation is provided. Gate count estimates for some of the architectures and tables for performing these estimates are presented. These tables are critical to future design work. Finally, a completely systolic architecture for an adaptive beamformer tracking system is developed.

1.1 Scope of Work

An adaptive algorithm study for array signal processors has been performed. The basic approach was to study the hardware/software tradeoffs of conventional (Von Neumann) and non-Von (parallel, pipeline, vector, array, and custom processors) with non-conventional arithmetic (SBNR) in an attempt to identify optimal algorithms (of order area x time) which are computationally fast yet flexible. A two step process was assumed; first, the sequential algorithms were speeded-up (seeking inherent parallelism) and second, fast algorithms were mapped onto new VLSI architectures (via recursion and pipelining). The purpose of this study is to provide theoretical design tools and interconnection strategies capable of achieving real-time implementation of signal processing algorithms via limited user-programmable mechanisms (e.g., firmware). Flexible firmware-oriented architectures dedicated to signal processing can then be identified. A systolic array for performing recursive least-squares minimization is proposed. It performs an orthogonal triangularization of the data matrix using a pipelined sequence of Givens rotations and generates the required residual without having to solve the associated triangular linear system by back-substitution.

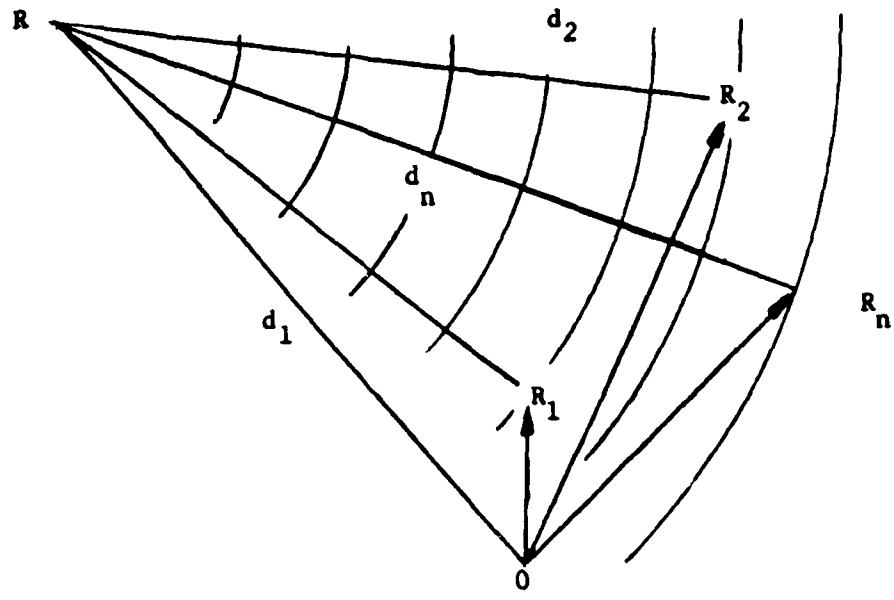
1.2. Current Efforts

An investigation of various systolic array architectures was undertaken. At the extreme ends were the GAPP array and the WARP array. WARP utilizes 68000 microprocessors in each processing element (PE). GAPP uses a 1 bit ALU with 128 bit RAM as each PE. Although more primitive (68000 is a 16 bit parallel engine), GAPP is a single chip of 72 PE's. Very large single chip upgraded GAPP's are in development at NCR. Because of its high speed and availability, GAPP is currently a superior choice. Between these outlying architectures were conventional and distributed arithmetic-based architectures.

A study of basic PE's was made. Because redundant number systems make fault tolerant computing easy, a new PE utilizing a primitive cell suggested in this report is proposed for a systolic array PE. Section 7.0 describes investigations of this novel PE. A PE based on a distributed arithmetic cell is studied. This cell increases computation speed by reducing multiplication to table-lookup of partial products and a series of shift/add operations. Whatever choice of PE is made, a systolic array will be feasible in current VLSI technology only if the VLSI model in section 4.0 is adhered to. Basically, the design constraint requires that communications be "nearest-neighbor" only. The control unit implementation for systolic arrays in which the microcode realizes the recursive LS algorithm has been studied. A design which permits high level algorithmic control constructs (for, while, if statements) to be efficiently mapped is discussed in Section 9.0.

1.3 Beamforming Architectures

An antenna beam is a collection of point sources or receptors (transmitter or receiver, respectively). The geometry governs the characteristic equations of the system. Figure 1 depicts an arbitrarily spaced array which is now used for discussion. Assume a coordinate system where R_n is a vector from the origin of the coordinate system to the nth array element. R is a vector from the origin to the assumed source location. Using this notation, we can describe the delays required for a delay-and-sum beamformer which are proportional to the distances from the source to the various elements as $d_n = ||R-R_n||$.



$$d_n = ||R-R_n|| \approx ||R|| - |U,R_n|$$

WHERE $U = R/||R||$

Figure 1 Propagation Geometry for Nonuniformly Spaced Array

Without loss of generality, let g_n denote the complex amplitude of an incoming signal at the nth array element. For discussion purposes we let the signal be decomposed via a temporal Fourier transform. Hence we must only form beams separately for each frequency. This merely simplifies matters for discussion and does not restrict analysis. In the far field, the desired beamformer output is given by (1) where **U** is a unit vector in the same direction as **R** and λ is the wavelength corresponding to a signal component frequency.

$$\sum_{n=0}^{N-1} g_n \exp(-j2\pi[U,R_n]/\lambda) \qquad (1)$$

Substituting for **U** and R_n, we have the following equation as a function of the incidence angle, a.

$$G(a) = \sum_{n=0}^{N-1} g_n \exp(-j2\pi(x_n \cos(a))/\lambda \qquad (2)$$

For the special case of a uniformly spaced line array as depicted in Figure 2, (2) reduces to the following form:

$$G(a) = \sum_{n=0}^{N-1} g_n \exp(-j2\pi(nd \cos(a)/\lambda) \quad (3)$$

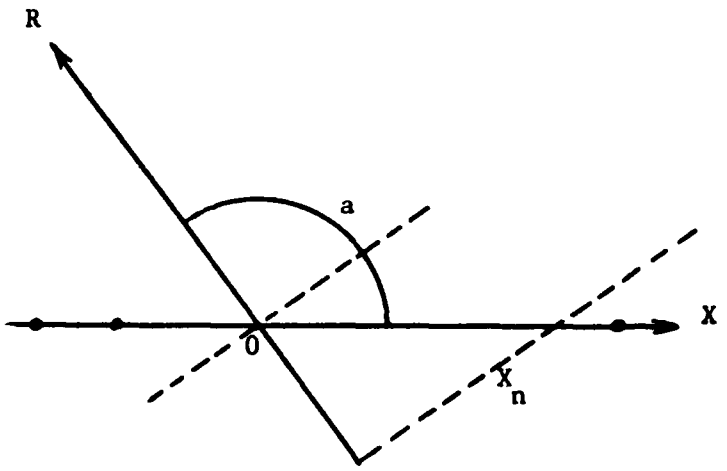
Equation (3) has the basic form of a DFT. When we consider that $\cos a_k = (k/N)(\lambda/d)$, the beamformer output at angles a_k is computable by the DFT as follows.

$$G_k = \sum_{n=0}^{N-1} g_n \exp(-j2\pi nK/N) \quad (4)$$

From this we can easily see that a 2-D temporal-spatial Fourier transform can form beams in the nonuniformly spaced look directions given by:

$$a_k = \cos^{-1}(k\lambda/nd) \quad (5)$$

Hence, the DFT plays a significant role in beamforming. Adaptive beamforming must then cause the beam pattern to favor certain spatial and/or spectral parameters.



$$|R_n, U| = X_n \cos a$$

$$G(a) = \sum_n g_n e^{-i2\pi f X_n (\cos a)/c}$$

FOR A UNIFORM ARRAY, $X_n = nd$

$$G(a) = \sum_n g_n e^{-i2\, n\pi f\, d\, (\cos a)/c}|$$

Figure 2 Line Array Beamforming

2.0 Least-Squares Minimum - Problem Definition

Given an n x p matrix X(n) and an n-element vector $\underline{y}(n)$ the corresponding n-element least-squares residual vector is defined by

$$\underline{e}(n) = X(n)\underline{w}(n) + \underline{y}(n) \tag{6}$$

where $\underline{w}(n)$ is the p-element vector of weights which minimizes

$$E(n) = ||B(n)\underline{e}(n)|| \tag{7}$$

and $||\cdot||$ denotes the usual Euclidean norm. Assuming the notation

$$\begin{aligned} X(n) &= [x_1, x_2 \ldots x_n], \\ \underline{y}(n) &= [y_1, y_2 \ldots y_n], \text{ and} \\ \underline{e}(n) &= [e_1, e_2 \ldots e_n] \\ n &= 1, 2, \ldots \end{aligned} \tag{8}$$

the iterative least-squares problem may be stated as follows. For successive values of n = p, p+1 ... evaluate the least-squares residual

$$e_n = x_n w(n) + y_n \tag{9}$$

The diagonal matrix

$$B(n) = \text{diag}\{b^{n-1}, b^{n-2} \ldots\ldots 1\} \tag{10}$$

has been included in equation (7) for increased generality. It applies an exponential weight factor b^{n-k} $(0<b<1)$ to each row x_k of the matrix X(n) and this has the effect of progressively weighting against the preceding rows of X(n) in favor of the nth row whose weight factor is unity. The more conventional unweighted least-squares problem is obtained by setting b = 1 in which case B(n) becomes a simple unit matrix.

For any value of n (> p) the least-squares problem defined above may be solved by the method of orthogonal triangularization (QR decomposition) which is numerically well-conditioned and may be described as follows. Generate an n x n unitary matrix Q(n) such that

$$Q(n)B(n)X(n) = \begin{bmatrix} R(n) \\ \hline o \end{bmatrix} \tag{11}$$

where R(n) is a p x p upper triangular matrix. Then, since Q(n) is unitary we have

$$E(n) = ||Q(n)B(n)e|| = \left|\left| \begin{bmatrix} R(n) \\ \hline o \end{bmatrix} \underline{w}(n) + \begin{bmatrix} \underline{u}(n) \\ \hline \underline{v}(n) \end{bmatrix} \right|\right| \tag{12}$$

where

$$\underline{u}(n) = P(n)B(n)\underline{y}(n) \tag{13}$$

and

$$\underline{v}(n) = S(n)B(n)\underline{y}(n) \tag{14}$$

P(n) and S(n) being the matrices of dimension p x n and (n-p) x n respectively which partition Q(n) in the form

$$Q(n) = \left|\begin{array}{c} P(n) \\ \hline S(n) \end{array}\right| \tag{15}$$

It follows that the least-squares weight vector $\underline{w}(n)$ must satisfy the equation

$$R(n)\underline{w}(n) + \underline{u}(n) = 0 \tag{16}$$

and hence

$$E(n) = ||\underline{v}(n)|| \tag{17}$$

Since R(n) is upper triangular, Equation (16) may readily be solved by a process of back-substitution and the resulting weight vector $\underline{w}(n)$ could obviously be used to evaluate the iterative least-squares residual defined in equation (9).

Adaptive processing can be defined as follows: The signals from the sensors are processed by filters whose outputs are linearly combined as in Figures 3a and 3b. A direction of observation is fixed as for conventional beamforming. The transfer functions of the filters $h_k(f)$ are determined in order to minimize the spectral density $Y_y(f)^k$ of the processor output signal y(t), with the constraint that the signal s(t) received from a possible source in the look direction be undistorted at the processor output (which is identical to keeping constant the transfer function of the array processing in the look direction). In order to be able to apply adaptivity, it is necessary that the shape of the wavefront received from a source as seen by the receiving channels be a known function of the source position.

That definition of the processing leads to the optimum filtering vector given by the expression

$$H_{opt}(f) = S^{-1}(f)d(f)[d^T(f)S^{-1}(f)d(f)]^{-1}$$

$H_{opt}(f)$ is a column vector composed of the K transfer functions $h_k(f)$. $S(f)$ is the (KxK) spectral density matrix, composed of the cross-spectral densities between the signals from every pair of receiving channels. d(f) is a column vector composed of the K transfer functions between a source in the look direction and the receiving channel output, normalized by the

transfer function between the source and a reference channel.

This optimum filtering vector is of course unrealizable in practice because the matrix S(f) is unknown. Adaptive array processing is a processing which uses an estimate of that filtering vector. It can be obtained by various procedures which can be classified into two groups: **direct** or **closed-loop methods.** In the first group, there are methods that estimate the matrix S(f) and use its inverse in the above equation, instead of the inverse of S(f). The second group is composed of the methods that directly compute the filtering vector with an iterative optimization algorithm, which avoids the estimation of S(f) and its inversion. It is this last method we are investigating.

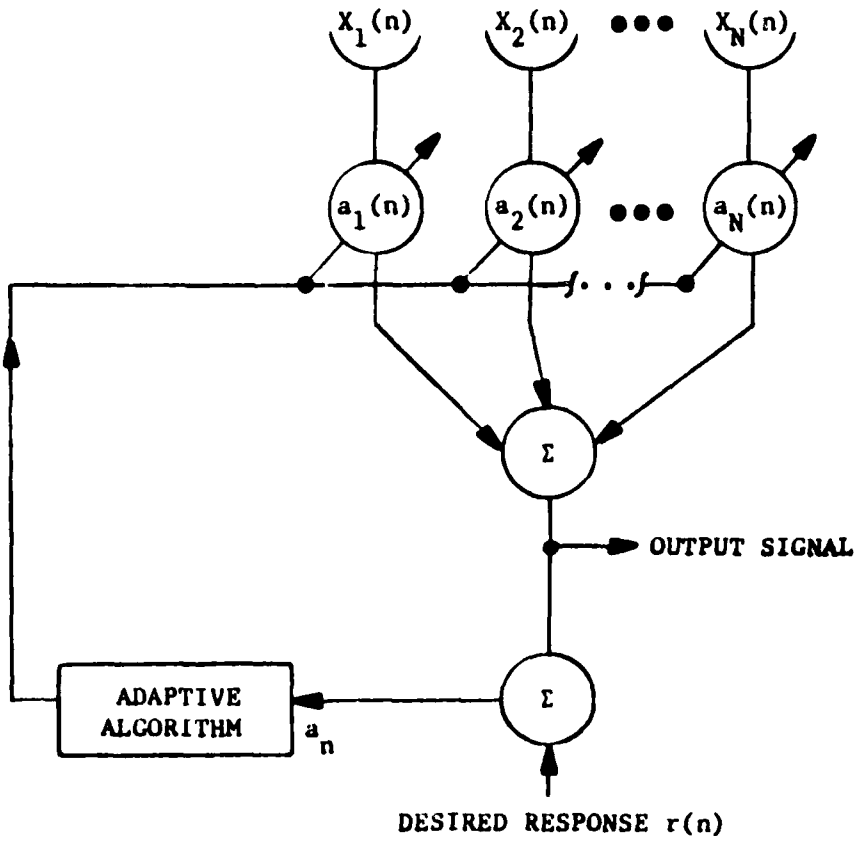


Figure 3a Adaptive Antenna

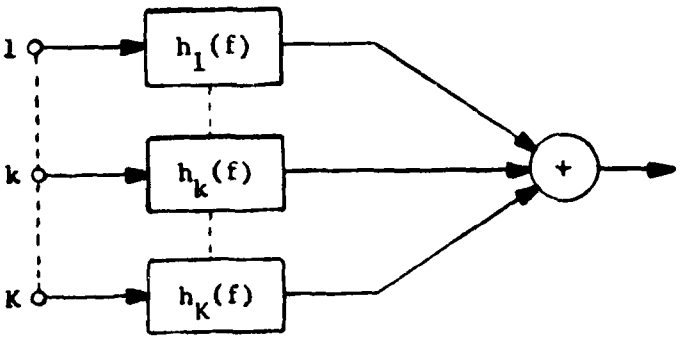


Figure 3b Spatial Filter Scheme

3.0 Solution Space

Assuming that a VLSI solution is sought, current fabrication technology insists that:

a. Processor interconnections are planar.

b. Processor interconnections are nearest neighbors only.

c. Processing elements are primitive.

d. Only localized data-flow is possible.

e. Pipelining data and/or tasks are desirable.

f. Maximum throughput requires evenly spaced computational load-shedding.

Given the kinds of mathematical tasks for adaptive beam communication antennas, methods now described do not suffice for the following reasons:

1. Householder - Golub Factorizations [1]:

An inflexible architecture [2] is required to handle the required global data communication.

2. Normal Equations:

Because squaring a matrix can become an ill-conditioned task, such methods rely on computationally burdensome orthogonal transformations.

3. Standard Givens Rotations:

Althcugh easily adapted to systolic arrays [3], some methods still require a square root computation per transformation. This translates into a severe computational bottleneck.

4. Fast Givens Rotations:

If the matrix is in factored form, no square roots are needed and half as many products are required. Still, nearest neighbor pivoting increases computational complexity and circuit area. Finally, under and overflow prevention are awkwardly handled [4].

An appropriate systolic array solution is a set of Scaled Givens rotations (SR), which do no pivoting and square-rooting. One (SR) scheme has been investigated and is reported in Section 8. (As will be seen eventually a weight matrix $W=D^2$ (not identity) easily achieves the necessary fading memory requirements often found in signal processing and least-squares filtering.)

4.0 VLSI Model

Our VLSI model of computation to derive a complexity measure is based on the following generally accepted assumptions [5,6,7]:

a. Wires have minimal width P=A(const); hence P^2 is the unity of measure for the area.

b. The area required to store one bit of information is $A(P)^2$; the distance between parallel wires is A(P).

c. Double layer metalization is allowed.

d. Wires run only horizontally and vertically.

e. Each transistor needs a minimal transition time, Y=A(k), k is a constant, to change its state. Thus Y is the unit execution time.

f. A binary signal propagates along a wire in time A(Y). Any long wires of length, L, require respective buffer/drivers with area A=A(P) x O(L).

5.0 Systolic Array Architecture

Systolic arrays may be configured as shown in Figure 4. They include rectangular, hexagonal, or linear systolic arrays. The most likely use for each configuration is also indicated. In the problem setting of adaptive beamforming, it is suggested by many researchers that the triangular array is preferable. Unfortunately, all commercially available systolic arrays including the NCR GAPP (Geometric Arithmetic Parallel Processor), incorporating 72 PE's (processing elements) are configurable in rectilinear not triangular fashion.

Hence, a triangular array configuration although optimal from an algorithmic standpoint does not efficiently utilize commercial arrays. This is an important remaining research issue in this study. Namely, how should rectilinear systolic arrays support recursive least-squares minimization computations utilizing all of the PE's instead of less than one-half?

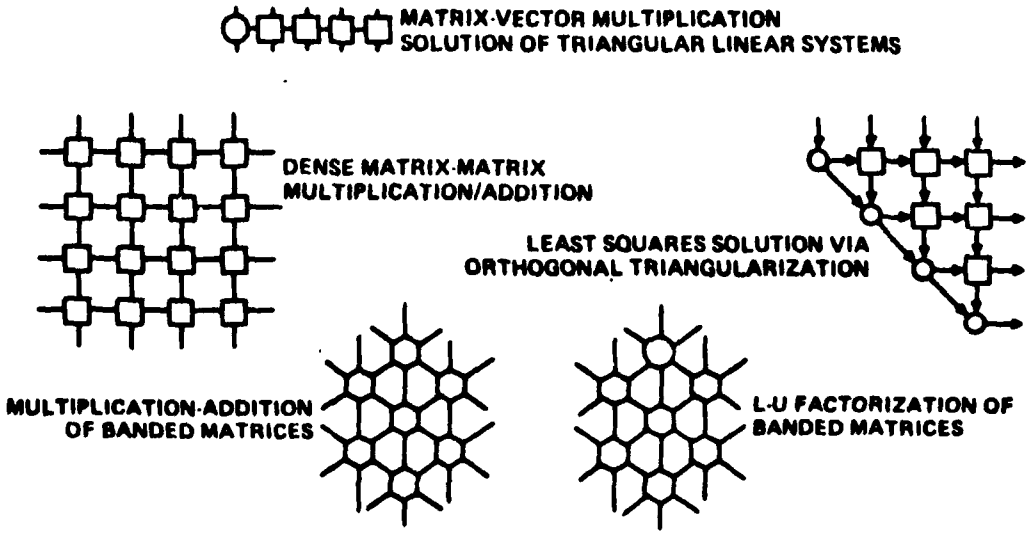


Figure 4 Systolic Array Solutions

The Fujitsu array shown in Figure 5 is a commercially available sample of the WARP architecture of 68000 microprocessors. Constructed mainly for parallel processing of images, it has 64 cells of Intel 80186 microprocessors. This array is not chip level integration but rather board level integration. As such it is highly unlikely that a system clock faster that 1 MHZ is possible. Hence, it is 10 times slower than a GAPP systolic array (clockwise only).

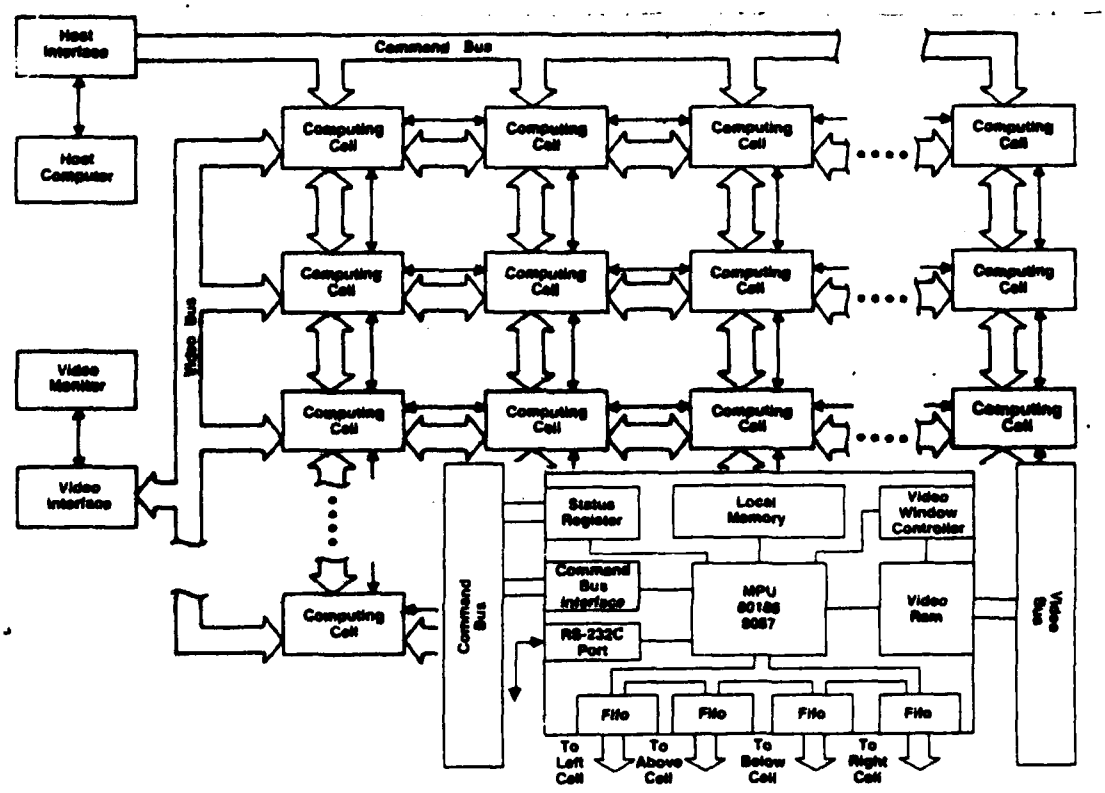


Figure 5 Fujitsu Array

6.0 Elementary GAPP Operations

Because almost all LS algorithms are multiplication intensive, an analysis of matrix x matrix multiplication on the GAPP was undertaken. Table 1 is a comparative analysis of speed versus wordlength for three GAPP systolic arrays.

The array size represents the number of processing elements (PE's). Furthermore, the tabulated data assume that a one-to-one mapping from matrix element to PE is provided. Hence, a 192 x 192 array size specifies a matrix x matrix multiplication in which each matrix has 192 rows and 192 columns. The systolic array size is also 192 x 192 PE's. Note especially that 16 bit integers in a 192 x 192 matrix by matrix multiplication require only 6.83 nsec per multiply.

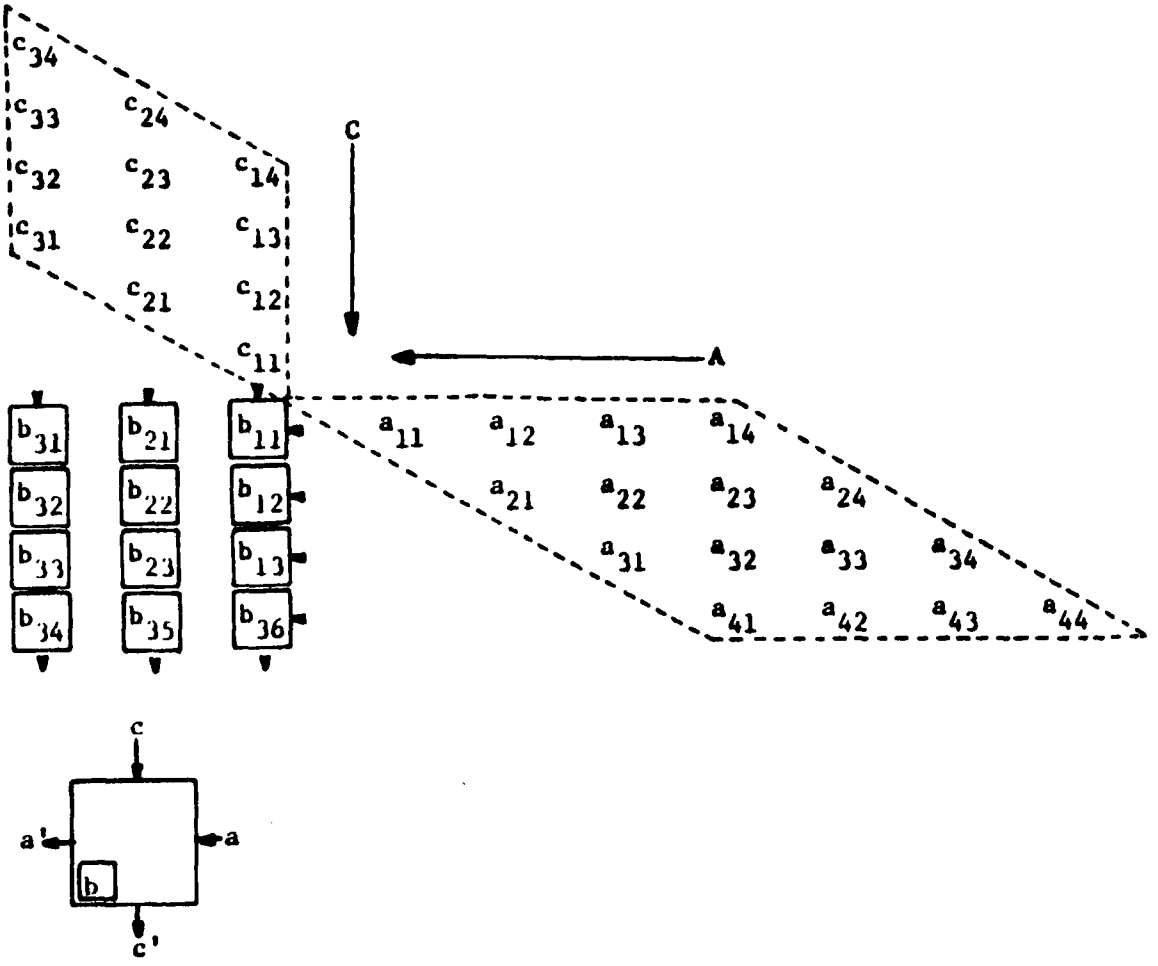
Table 1 Speed Estimates for M x M Matrix Multiply

ARRAYSIZE	192 X 192	504 X 504	1008 X 1008
Chip Count	512	3,528	14,112
Current Cost	$76,800	$529,200	$2,116,800
# operations	14,118,912	255,794,112	2,047,368,960
8 bit integer			
# Cycles	261,120	685,440	1,370,880
# Cycles/op	0.0185	0.0039	0.0007
ns/op	1.85	0.39	0.07
16 bit integer			
# Cycles	964,608	2,532,096	5,064,192
# Cycles/op	0.0683	0.0099	0.0025
ns/op	6.83	0.99	0.25
32 bit integer			
# Cycles	3,698,688	9,709,056	19,418,112
# Cycles/op	0.2607	0.038	0.0095
ns/op	26.07	3.8	0.95

The actual GAPP Algorithmic Language (GAL) code to generate the tabulated data is found in Figure 6a. This "C"-like code invokes a bit-parallel word-parallel scheme as proposed in [8]. The procedure assumes that one matrix is resident in the systolic array, that the other matrix is skewed into the array from the east and that the product matrix exists in skewed fashion to the south. The matrix-matrix multiplication array is found in Figure 6b. The purpose of this exercise is to obtain real-time speed performance ratings on a 10 MHZ commercially available systolic array.

```
/*This program performs a magnitude multiplication*/
/*Variable dictionary:
                        A - multiplier
                        B - multiplicand
                        C - result
                        D - local temporary storage

  Algorithm:  And the multiplier with each bit of the multiplicant. After
              each multiplicant bit is finished operating on the multiplier
              add the temporary result to the result RAM using a shifted
              add.

*/

image A:0:7;     /*multiplier*/
image B:8:15;    /*multiplicand*/
image C:16:31; /*result*/
image D:32:39;

main()
{
        int m = size(A);
        int n = size(B);
        int i,j;


        for(i = 0; i < n; i++)                  /*For each bit of B*/
        {
                B:i ew:=ram;

                for(j =0; j < m; j++)           /*For each bit of A*/
                {
                        A:j ns:=ram c:=0;       /*And the bits*/
                            c:=cy;
                        D:j ram:=c;             /*Store the result in D*/
                }

                c:=0;
                for(j = 0; j < m; j++)          /*Add the shifted temporary
                                                  result to C*/
                {
                        D:j       ns:=ram;
                        C:(j + i) ew:=ram;
                        C:(j + i) ram:=sm c:=cy;
                }
                C:(j + i) ram:=c;
        }
}
```

Figure 6a Typical GAPP Matrix Multiply Routine



a' <- a

c' <- c + ab

Figure 6b Matrix x Matrix Multiply Array Using Static Coefficients

7.0 SDNR - Systolic Arrays

7.1 Redundant Numbers

In the most general sense, a redundant number system allows both an increase in the number of positive digits and negative digits as follows.

Definition 1.

$$w_{red}^{(n,m)} : R \dot{x} R \dot{x} \ldots \dot{x} R \dashrightarrow Q \tag{18}$$

$$a_{n-1} \cdots a_0 \, a_{-1} \, a_{-2} \cdots a_{-m} \dashrightarrow \sum_{i=-m}^{n-1} a_i \, d^i \tag{19}$$

where the digits $d_i \in R := \{-r_1, -r_1+1, \ldots, 0, 1, \ldots, r_2-1, r_2\}$ (20)

$$(r_1, r_2 > 0)$$

The representation described by Eqs. (18), (19), and (20) is called **redundant notation with base** d. The above mapping of the number representation (or notation), w_{red} assigns to a sequence $a_{n-1} \cdots a_{-m}$ of digits a value from a range Q where Q may be an integer, the reals, or zero.

A notation is redundant if there is at least one $q \in Q$ which is represented in more than one way. That is, if there are sequences (apart from zero)

$$a_{n-1} \cdots a_{-m} + b_{n-1} \cdots b_{-m} \dashrightarrow w^{(n,m)}(a_{n-1}, \ldots a_{-m}) = w^{(n,m)}(b_{n-1}, \ldots b_{-m}) \tag{21}$$

However, the general redundant representation does not lead to efficiencies in algorithm computations or implementations. As we shall see shortly, if the following restrictions are placed upon the general redundant notation, very attractive properties support efficient implementations.

The basic properties of signed-digit number representations (SDNR) are:

a. The radix, d, is a positive integer.

b. The SDNR of the algebraic quantity, zero, is unique if

$$m=n, \; (d-1) \geq m \quad \text{and} \quad m-1 \geq (d-1)/2 \tag{22}$$

c. Transformations between conventional representations and SDNR exist.

d. Totally parallel addition/subtraction are possible.

e. Addition and subtraction of two numbers are free of serial propagations of carry/borrows.

f. SDNR numbers are positionally weighted.

g. The polarity of an SDNR number is given by the polarity of its most significant non-zero digit.

h. No special treatment is needed for the most significant position.

i. Addition/subtraction time is independent of operand length.

Avizienis [9], Atkins [10], Tung [11], Ercegovac [12], and Robertson [13] have shown that SDNR can effectively operate in a general purpose digital computer for the following reasons.

1. Redundancy introduced into the adder-subtractor structure reduces (but does not entirely eliminate) carry-borrow propagation leading to rapid multiplication.

2. Full precision comparison of the divisor and partial remainder in division algorithms is not required because quotient digits can be determined from relatively few high order bits.

3. Negation is a simple logical complementing of the sign bits (e.g., unlike two's complement notation which requires an additional step, adding an LSB "one"). As was seen in the ILLIAC III [10], such negation expedites execution of floating point addition and subtraction.

4. Variable length operand formats and parallel vector arithmetic are facilitated by basic properties of SDNR's. First and foremost, operations can proceed from left-to-right (rather than right-to-left as required in 1's, 2's complement representations). Secondly, if appropriately implemented, the position of the least significant digit need not be known for address and subtractors.

5. Because a signed-digit combination adder/subtractor needs no carry/borrow in the LSD, the ALU can be partitioned into identical and cascadable single digit adder/subtractors. VLSI implementations tend to become highly regular.

6. Multiplication with SDNR tends to automatically produce rounded results (of great importance in computationally intensive signal processing applications). In fact, Robertson, based on work by Rohatsch [14], has shown that the probability of obtaining a rounded result is 5/6.

7. SDNR allows unusual algorithms such as wired-in significant-

digit arithmetic [15] and dual notation algorithms capable of accepting both SDNR and conventional operands (1's, 2's complement) to produce SDNR results [16]. These observations lead to the implementation of a universal Arithmetic Building Element (ABE) capturing not only the preceding algorithms but also efficiently separating functions of logic designs and arithmetic design [17].

8. Overflow detection can occur immediately following the production of most significant result digits (unlike conventional notation).

7.2. Efficient SDNR Realization

Several implementations based on the SDNR have already been investigated [18,19,20,21]. All of these, however, sought to satisfy general data processing requirements of a mainframe computer. In contrast, signal processing applications are generally multiplication/addition intensive. Of late, the utility of distributed arithmetic [22,23,24] has shed new light on bit-wise algorithms, which are essentially partial product and accumulation intensive. In the work reported here, it is assumed that the adaptive signal processor must handle additions and multiplications rapidly.

An allowed digit set (-1,0,1) which is a subset of the SDNR is assumed. A redundant signed binary number representation (SBNR):

$$X_n X_{n-1} \ldots X_1 \dashrightarrow X_i \in (-1,0,1) \tag{23}$$

represents a number whose value is expressed as

$$\sum_{i=1}^{n} X_i \cdot 2^{i-1} \tag{24}$$

The importance of SBNR is as follows.

a. Conversion of unsigned binary numbers to SBNR is unnecessary as they are identical.

b. Since a two's complement binary representation $(X_n X_{n-1} \ldots X_1)_{\overline{2}}$ expresses the number

$$-X_n 2^{n-1} + \sum_{i=1}^{n-1} X_i \cdot 2^{i-1} \tag{25}$$

This same number can be expressed in SBNR by

$$(X_n X_{n-1} \ldots X_1)_{SD} \tag{26}$$

because the sign bit X_n in two's complement representation is considered to have weight -2^{n-1}. Hence, conversion from

signed binary two's complement representation to SBNR is simply an inversion of the sign bit alone!

Avizienis [9] further demonstrated that the SBNR (radix d=2 with digit values -1,0,1) with a decreased redundancy requirement (invoking a two-step addition by allowing the propagation of the transfer digit over two digital positions to the left) requires only d+1 sum digits. In general, he showed that the lower limit of required redundancy of one digit depends on the number of digital positions the transfer digits propagate as follows.

If GIVEN:

a. no redundancy utilized

b. s_i, sum digit {d values only}

THEN:

$$s_i = f(z_i, y_i, z_{i+1} \cdots, z_m, y_m) \tag{27}$$

z_i = ith addend digit

y_i = ith augend digit

If, however, $s_i \in$ {d+1 values}, then Eq. (10) becomes

$$s_i = f(z_i, y_i, z_{i+1}, y_{i+1}, z_{i+2}, y_{i+2}) \tag{28}$$

and if $s_i \in$ {d+2 values or more}, then (10) becomes

$$s_i = f(z_i, y_i, z_{i+1}, y_{i+1}) \tag{29}$$

Using these observations, a single cell can implement the one digit adder/subtractor if certain choices for a redundant digit are always made. Specifically, let any redundant binary digit be represented by two bits X_s and X_d as follows where $\overline{1} = -1$.

Table 2 Redundant Digit Selection Rule

Redundant Digit X	Representation Sign X_s	Digit X_d
0	0	0
1	0	1
$\overline{1}$	1	1

Invoking this TRIT realization for our SBNR further simplifies the cell implementation without sacrificing the transfer digit propagation advantage. Using this subset allows six types of intermediate results in

the first of two addition steps as defined in Table 3.

Table 3 Intermediate Addition Step Classes

Type	Augend (x_i)	Addend (y_i)	Next Lower Position (x_{i-1}, y_{i-1})	Carry (c_i)	Intermed. Sum (s_i)
1	1	1	---	1	0
2	1	0	Both are positive	1	$\bar{1}$
	0	1	At least one is negative	0	1
3	0	0	---	0	0
4	1	$\bar{1}$	---	0	0
	$\bar{1}$	1	---	0	0
5	0	$\bar{1}$	Both are positive	0	$\bar{1}$
	$\bar{1}$	0	At least one is negative	$\bar{1}$	1
6	$\bar{1}$	$\bar{1}$	---	$\bar{1}$	0

The second step in an addition cycle adds s_i and c_{i-1} from the next lower position to obtain a sum digit z_i with no carry/borrow generation required.

If we allow any redundant binary digit to be represented as $X_s X_d$ with the redundant digit selection rule as prescribed in Table 2, the boolean equations which govern selection and addition per Table 3 are found in Eqs. (30), (31), (32), and (33). Two critical observations are made. The ith SBNR carries, C_s and C_d, depend only upon the ith, i-1 digits and i-1 carries. Hence, carry propagation extends only into the next adjacent digit column. SBNR addition does not require full-word carry propagation as in binary addition. SBNR addition makes systolic array implementations straightforward. Pre-scrambling bits or words is not required.

Secondly, the ith sum digits, Z_s and Z_d, depend only upon ith, i-1 operand digits and i-1 carries. Hence, just as for carry propagation, result digits achieve the same advantage in systolic array implementations.

$$Z_{d_i} = C'_{s_{i-1}}[(x_{s_{i-1}} + y_{s_{i-1}})\{x_{s_i} y'_{d_i} y_{s_i} + y_{s_i} x_{d_i} x'_{s_i} + x_{d_i} y'_{d_i} y_{s_i} + y_{d_i} x'_{d_i} x'_{s_i} + C_{d_{i-1}}\} + C_{d_{i-1}}(y_{d_i} x_{d_i} + y_{s_i} x_{d_i} + y_{d_i} x_{s_i} + x'_{s_i} x'_{d_i} y'_{d_i} y'_{s_i})] \quad (30)$$

$$Z_{s_i} = C_{d_{i-1}}[(x'_{s_{i-1}} + y'_{s_{i-1}})\{x_{s_i} y'_{d_i} y_{s_i} + y_{s_i} x'_{d_i} x'_{s_i} + x_{d_i} y'_{d_i} y'_{s_i} + y_{d_i} x'_{d_i} x'_{s_i} + C_{s_{i-1}}\} + C_{s_{i-1}}(y_{d_i} x_{d_i} + y_{s_i} x_{d_i} + y_{d_i} x_{s_i} + x'_{s_i} x'_{d_i} y'_{d_i} y'_{s_i})] \quad (31)$$

Intermediate Sum (Z_{d_i}) and Sign (Z_{s_i}) Digit Expansions

$$C'_{s_i} = (x'_{s_i} + y'_{d_i})(x'_{s_{i-1}} y'_{s_{i-1}}) + x'_{s_i} x_{d_i} + x'_{s_i} y'_{s_i}$$
$$+ y'_{d_i} y_{s_i} x_{d_i}$$
$$+ y'_{d_i} x_{s_i} x_{s_i} \qquad (32)$$

$$C_{d_i} = (y'_{d_i} + x'_{d_i})(x_{s_{i-1}} + y_{s_{i-1}}) + y_{s_i} y'_{d_i} + x'_{d_i} y'_{d_i}$$
$$+ x_{s_i} x'_{d_i} \qquad (33)$$

Intermediate Carry (C_{d_i}) and Carry Sign (C_{s_i}) Digit Expansions

7.3 Primitive VLSI Processing Element

A primitive cell suitable for large VLSI arrays and especially for adaptive signal processors must have few interconnections beyond its nearest neighbors and must have very simple controls. VLSI arrays effectively function in a data-flow manner. Fortunately, many signal processing algorithms can be implemented with distributed or bit-serial arithmetic. Mactaggart et. al., [25] and others have shown that bit-serial implementations offer a highly regular design and lower power consumption than conventional arithmetic. The cell we propose for a VLSI adaptive signal processor is depicted in Figure 7. This cell implements the basic addition/subtraction steps of Table 3 using the SBNR of (-1,0,1) and the redundant digit selection rule of Table 2.

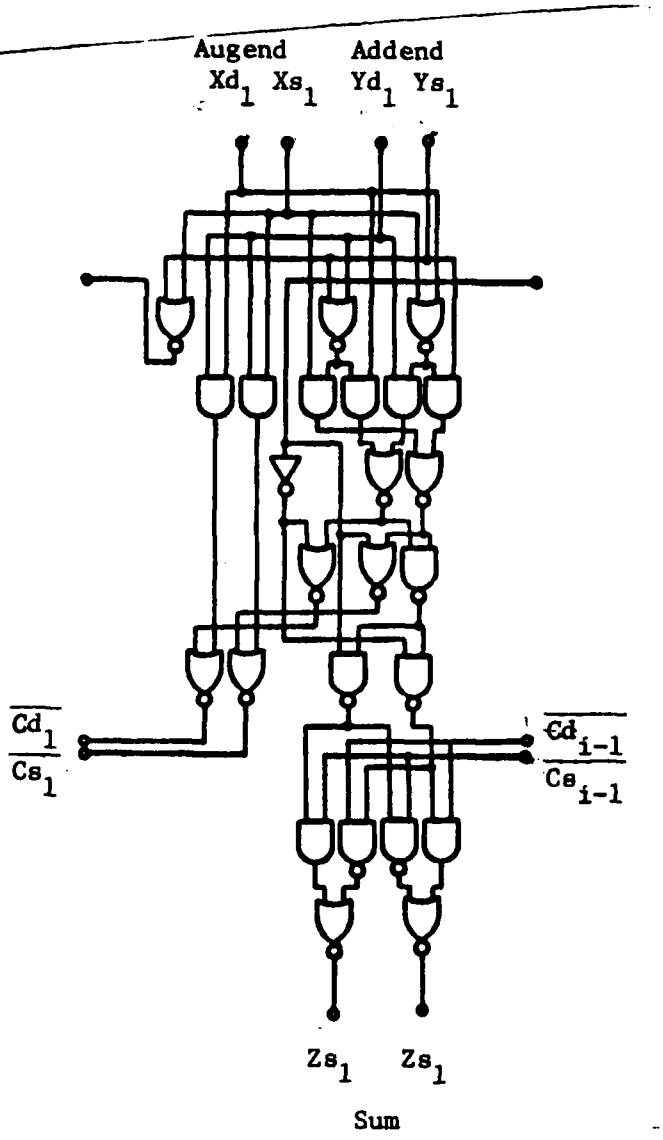


Figure 7 Primitive Bit-Serial Cell

The primitive SBNR adder cell of Figure 7 is only one of several possible implementations. For layout rule simplification, the circuits selected have used only 2-input gates. Hence, the logic depth/cell is not as optimal as it could be if n-input gates (n > 2) are allowed. Figure 8 represents the cell conceptually with dashed lines indicating the "data flow" internal to the cell. Subsequent discussions assume that our primitive SBNR cell is as shown in Figure 8 where North, South, East, and West (N,S,E,W) data paths are available. Inter-cell connections shall only be to the nearest neighbors. Furthermore, latches on the north and east are incorporated in the cell to aid systolic latching of operand bits. The cell is now utilized in a systolic array where the dominant operation of nxn matrix multiplication is invoked.

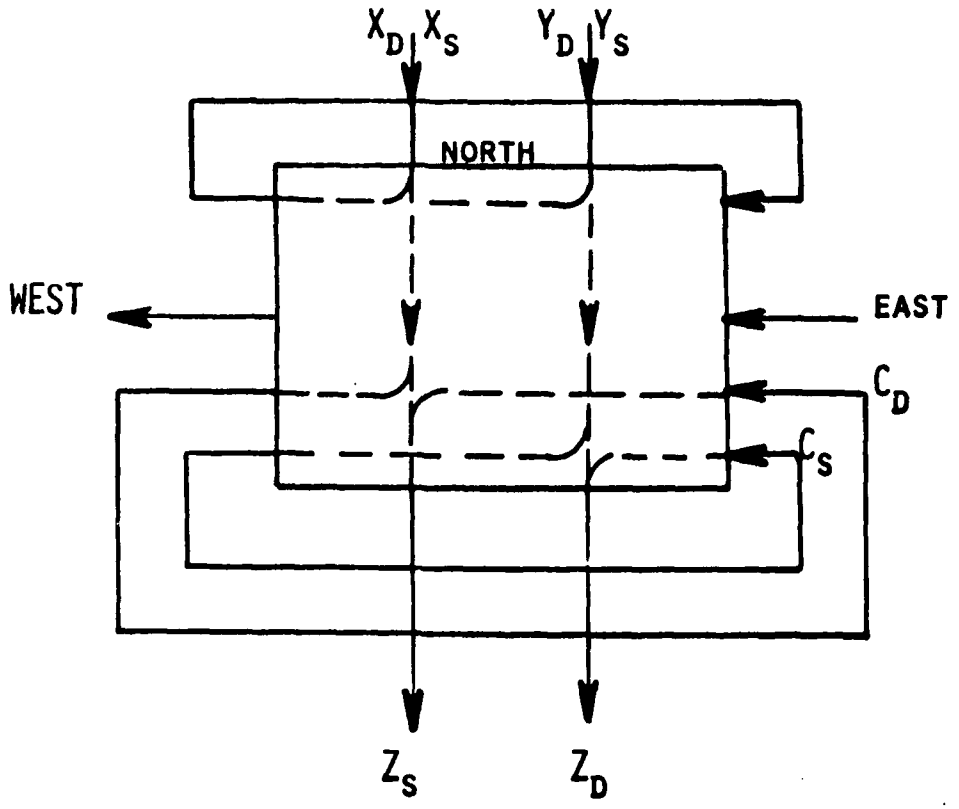


Figure 8 Primitive Cell (Internal Data Flow)

7.4 Matrix x Matrix Multiplication

Most of the computational effort expended by a digital signal processor is devoted to matrix x matrix multiplication. Such matrix operations may be either sums of word level products or sums of bit level products. We now know that a strong relationship exists between word and bit level systolic arrays [8]. If we treat such computational problems from the onset as bit level manipulations, fast area efficient VLSI arrays are possible [26,27]. In our implementations, a systolic-like bit level approach is assumed where each processing cell is a multiplier and gated full adder. However, the multiplier and adder utilize SBNR rather than two's complement arithmetic for reasons discussed earlier.

To understand these advantages, we discuss the implementations at the word level and show how the bit level similarities appear. Consider the multiplication of two n x n matrices, $\underline{S} = (s_{ik})$ and $\underline{H} = (h_{kj})$ to form the matrix product $\underline{Y} = (y_{ij})$ as in

$$y_{ij} = \sum_{k=1}^{n} s_{ij}\, h_{kj} \quad (i,j=1,2,\ldots,n) \tag{34}$$

Without any loss of generality, $\underline{Y}$ may be considered to be made up of independent vectors y_i. The aggregate of n matrix x vector product evaluations, each of the type

$$y_i = \sum_{k=1}^{n} s_{ik} \underline{h}_k \qquad (35)$$

comprises the matrix $\underline{Y}$. However, each of (2) is also an aggregate of n inner product evaluations of the type

$$y_i = \sum_{k=1}^{n} \underline{s}_k \underline{h}_k \qquad (36)$$

Multiplication of two matrices now becomes a series of unit multiplications of Eq. (36) and an accumulation of relevant product terms. Many systolic arrays use a multiplier/accumulator pair as the PE. But even Eq. (36) can be broken down further into a series of bit level sum of products. The coefficient of each power of two in the result now becomes a convolution of the coefficients in the two operands. We make use of this important discovery by organizing the input signal streams so that operations at the bit level are pipelined onto our array as follows. The tantamount constraint we must adopt is that the physical significance position in the array must be maintained so that partial products are accumulated correctly. Beyond that, we do not have to be concerned with the complicated carry/borrow strategies found in two's complement systems because our SBNR has a minimal carry/borrow distance (adjacent digit position only).

Consider the multiplication of two 4 x 4 matrices as in Figure 9. This diagram portrays the interaction of the bits of two sets of words, s_{11} and h_{k1}, which compute y_{11}. Each square is a gated full adder unit of cells depicted by Figure 8. The data words in Figure 9 have been expanded into their respective individual bits and the kth row is associated with the kth set of words. Words s_{1k} enter from the right while words h_{k1} enter from the left in a bit serial manner. Although we show the least significant bits (s_{1k},h_{k1}) entering ahead of the next significant bits (s_{11}, h_{k1}) with SBNR, the MSB's can enter first without any effect. As with all systolic arrays which compute products from both operands which move, bit level times are staggered by one clock cycle relative to the corresponding word on the row above. Upon forming partial products, the intermediate results, y_{ik}, are passed vertically downward. A chief advantage to SBNR now becomes clear. The carry/borrow bits generated by this process remain on fixed sites thus relieving data communication interconnections between cells.

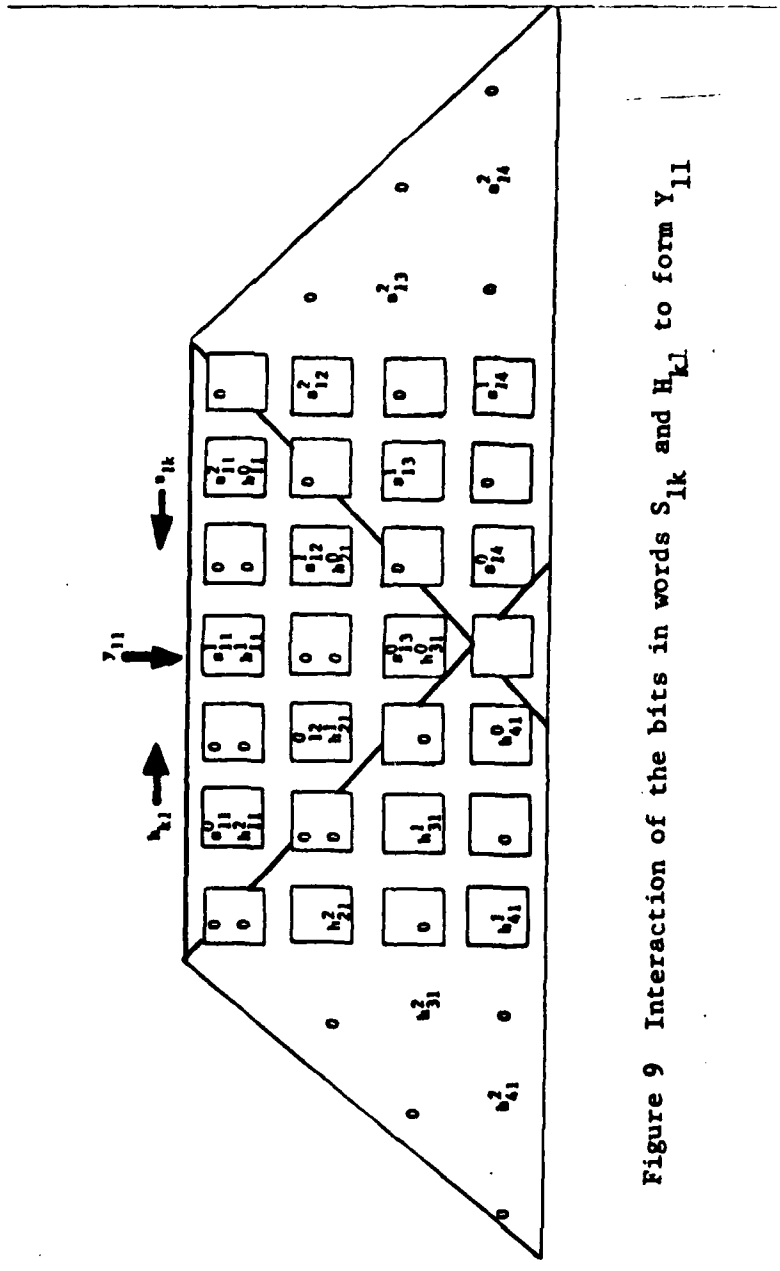


Figure 9 Interaction of the bits in words S_{1k} and H_{k1} to form Y_{11}

On a larger scale, the partial products y_{ij} are generated as in Figure 10 in the shaded areas. Dashed lines adjacent to the parallelogram edges are guardbands to allow for growth generated by carry bits. For m-

bit operands,

$$m + \lfloor 1/2 \log_2 n \rfloor \tag{37}$$

bits are necessary. These guard bands are equivalent to spacing input parallelograms with guardbands filled in with zero bits initially.

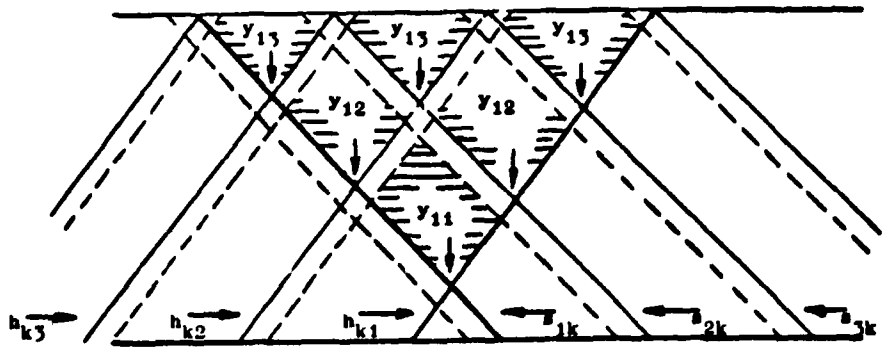


Figure 10 Partial Product Generation of Matrix x Matrix Multiplication

The shaded areas which move down vertically generate partial products such that successive cells at a given location in the shaded diamond area accumulate all terms in

$$y_{ij}^{pq} = \sum_{k=1}^{n} s_{ik}^{p} \, h_{kj}^{q} \tag{38}$$

Note how each row of cells within a diamond produces all the partial product sums required to form a bit of given significance in the result. The full sum of products is formed by the accumulation of diamonds emerging from the bottom. A pipelined tree of adder cells connected to the bottom edge generates the full sum which can be clocked out least significant bit first or most significant bit first (SBNR only). The full sum is then computed every 2m+1 clock cycles.

It is important to note that the symmetry of the diamonds in Figure 10 carry over directly into regular VLSI cells with few inter-cell connections, resulting in an extremely efficient VLSI computational array. If SBNR numbers are not used, carry/borrow logic and inter-cell data paths would be complicated by the same level of complexity necessary to fabricate full carry lookahead adders (where carry propagate logic grows as a function of word length). In an SBNR implementation, there is no need for distant cell paths. Nearest neighbor cell paths and replication of the primitive cell in Figure 8 are all that is required.

Another advantage to SBNR is the absence of special circuitry and algorithms to handle signal operands. In two's complement arithmetic, the Baugh Wooley algorithm can be used [28]. In this procedure, two's complement words are treated as positive numbers if

1. A fixed correction term is added to the result for each word level multiplication.

8.0 Systolic Array Least Squares Solution

McWhirter [29] proposes a set of 3 primitive cells arranged in a triangular systolic array which performs recursive least-squares minimization. Orthogonal triangularization of the data matrix is performed using a pipelined sequence of square-root free Givens rotations. The residual is generated without requiring solution of the resultant triangular linear system by backsubstitution. If desired the triangular system can be solved using an independent systolic array which performs the back substitution [30].

The square-root free Givens rotation triangular systolic array is shown in Figure 11. The associated primitive cells are given in Figure 12. To provide a common performance testbed, the conventional binary, SBNR, and distributed arithmetic architectures in Sections 10.0, 11.0, and 12.0 were studied based on an implementation of this systolic array.

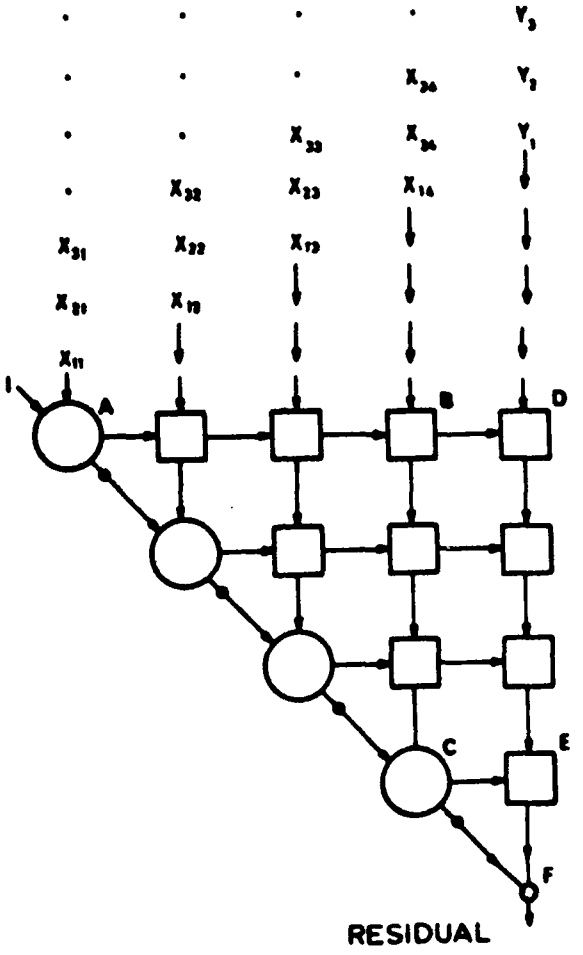


Figure 11 Systolic Array for Recursive Least-Squares Minimization

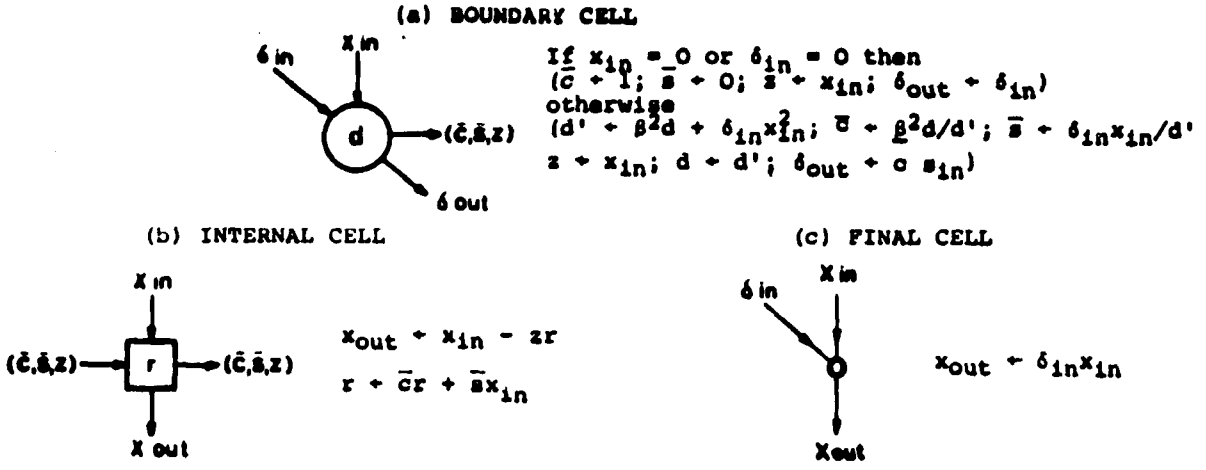


Figure 12 Cells Required for the Square-Root Free Givens Rotation

Barlow and Ispen [31] develop a scaled Givens rotation systolic algorithm. This algorithm is given in Appendix A. The scaled Givens rotation requires floating point processing. Since floating point processing algorithms have not been studied on the bit-serial architectures, actual performance estimates for the scaled Givens rotation are beyond the scope of this report. A qualitative comparison of the scaled Givens rotation versus the square-root free Givens rotation follows.

The scaled Givens rotation algorithm operates on banded matrices of width $w = p + q + 1$ where p is the number of superdiagonals and q is the number of subdiagonals. Assuming m rows in the banded matrix and z right hand side vectors, the number of computation steps are given by:

$$2m + 3(q+1) + z + 1 \qquad (41)$$

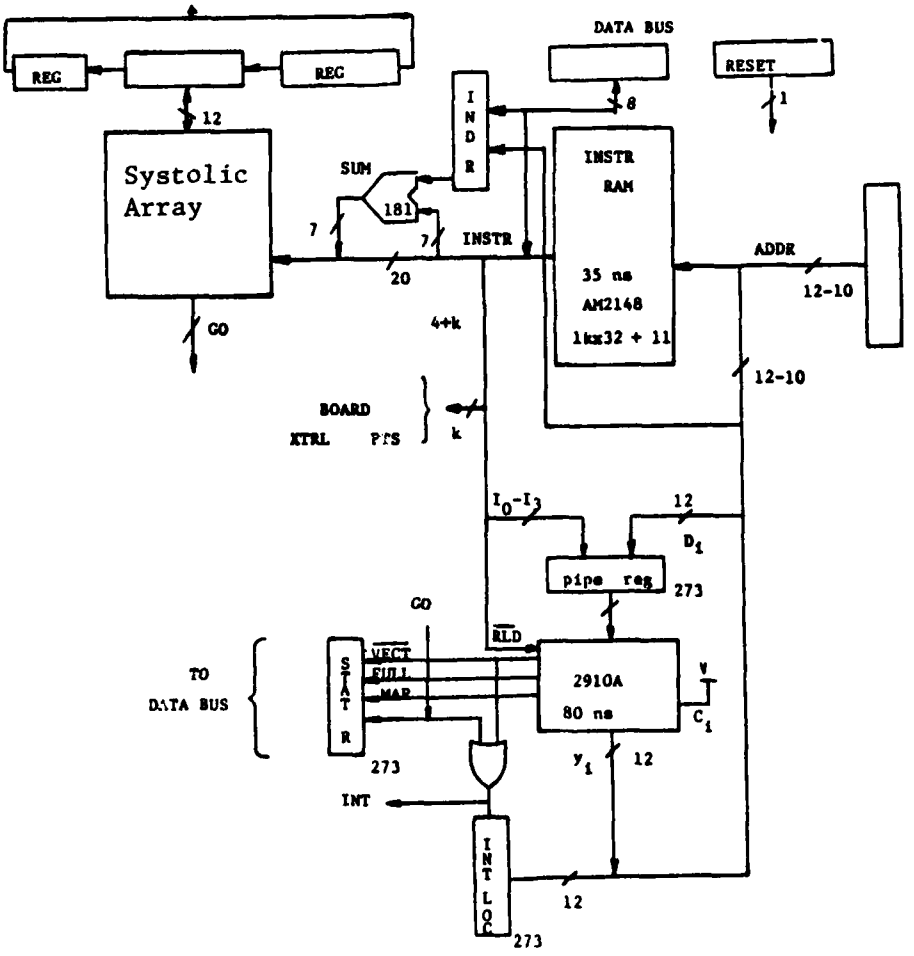
The individual cell complexity (the number of equations solved at each cell) is approximately the same as those for the square-root free Givens rotation. Only one division operation is required in the scaled Givens rotation and many of the multiplies are reduced to shift operations. This may save considerable computation time. Both scaled Givens rotations and square-root free Givens rotations have processor utilizations of approximately 50%.

A simulator package has been developed which allows simulation of a variety of systolic array algorithms. This package is presented in Appendix C and Appendix D. Appendix C is a listing of all the simulator generator code. Appendix D contains a simulation specification file for the square-root free Givens rotation. Appendix E demonstrates the square-root free Givens rotation by listing a sample run of the simulator on an example data matrix.

9.0 Control Unit Design

A suitable control unit design has been investigated (Figure 13). This architecture assumes a generic systolic array which may be composed of GAPP, SBNR, or distributed arithmetic processing elements. The major components of the control unit architecture are a rectilinear or triangular systolic array, an Am2910 microcode sequencer, an instruction RAM for control signal storage and generation, a RAM address register for systolic array addressing, and an adder for mapping high-level address computation schemes.

The control unit architecture is well suited to provide easy implementation of the Givens rotation algorithm. The addressing logic for the systolic array RAM is critical because a variety of address computations can be performed. Hence, address loops for bit sequencing are simplified. This control unit architecture is assumed in the systolic array architecture performance evaluations.



k = number of on-board control points (CC, clock xtrl, Sync, Int En, muxes, demuxes)

Figure 13 Control Unit Architecture

10.0 Conventional Binary Implementation

The GAPP™ is a commercially available systolic array device providing 72 conventional binary processing elements. GAPP is dimensioned as a 6 X 12 rectilinear array. The square-root free or scaled Givens rotation can be easily implemented on this device, although not optimally fast.

In order to obtain realistic speed estimates for a conventional binary implementation of the square-root free Givens rotation, code for the GAPP device has been written in a C-like language known as GAL™. This GAL code is found in Appendix B. GAL code is both high-level and low-level. Low-level microinstructions for GAPP control are stated explicitly by inclusion of mnemonics. Loop control and GAPP RAM address control are expressed in C-like constructs. Though these high-level statements can not be designated as microcode, they are readily mapped by hand or compiler to the control unit architecture given in Section 9.0.

Emulation of the square-root free Givens rotation systolic array requires definition of three distinct processor types; boundary, internal, and final cells. These processors form the triangular systolic array shown in Figure 11. Each processor type performs a distinct set of data computations. The equations defining each processor type are given in Figure 12.

Since GAPP is an SIMD machine it is impossible to operate all three processor types simultaneously on a single array. It is possible to create "masks" in the RAM space at each processor which control whether or not results of a computation are to be stored at a particular processing element's RAM. By producing masks which block the modification of RAM at a processor, only specific cells (i.e. boundary cell) will have any effect during a cycle of computation. Hence, it is possible to simulate the MIMD structure required for the Givens rotation.

The first sequence of operations in the GAL square-root free Givens rotation code performs a MOVE of all appropriate data to the correct processors. Then the subroutines for the boundary cell, internal cell, and final cell are executed in sequential order. Masks block the writing of results to RAM at all processor locations except those locations which emulate the currently operating cell type.

The GAL code processes fixed-point signed magnitude numbers. The radix is just right of the sign bit and precision is 8 bits. Scaled Givens rotation requires that floating point operands be used. Code exists to perform floating point operations but currently has not been publicly released by NCR. When this code becomes available, the scaled Givens rotation can be implemented with a minimum of effort.

GAPP and GAL are registered trademarks of NCR Corp., Ft. Collins, CO.

The GAL square-root free Givens rotation code requires approximately

$$(2r + c + 1)(83n^2 + 224n + 156) \quad (42)$$

instruction cycles where n is the bit length of the input operands, r is the number of matrix rows, and c is the number of matrix columns. The latency from first input to first residual output is

$$(c + r + 1)(83n^2 + 224n + 156) \quad (43)$$

The time to complete the entire matrix reduction increases linearly with the size of the array. The number of elements processed, however, increases as the square of the array size. Hence, there must be some point at which the processing power exceeds non-systolic methods. One way to estimate the processing power of the GAPP solution is to compute the number of array elements processed per instruction cycle. Assuming a fixed word length, the word length dependent term is a constant $k = 83n^2 + 224n + 156$. The number of array elements processed per cycle is:

$$rc/(2r + c + 1)k \quad \text{elements/cycle} \quad (44)$$

It can be seen that for square matrices ($r = c$) the number of elements processed per cycle increases quadratically as the array size increases. Improvements in speed may be obtained if concurrency can be achieved in the operation of the three cell types of McWhirter's algorithm. A promising solution is to use three separate arrays (one for each cell type), and carefully synchronize data flow between the arrays. This is a suggested topic for future development of the GAPP solution.

Another conventional arithmetic architecture to be considered is a cell containing sufficient binary multipliers and adders to achieve maximum concurrency in cell computations. Data dependencies among the equations to be realized must be considered when designing such an architecture.

The boundary cell of McWhirter's algorithm can perform at most three multiplies/divides simultaneously, given the data dependencies among the equations. A configuration with three multipliers and one adder would permit a cell to perform all the boundary cell operations in three cycles, where a cycle is the time for a multiplier to perform a word computation. An internal call can be constructed with three multipliers and two adders and perform all computations in two cycles. The final cell just requires one multiplier and one cycle of computation time.

The highly parallel processing of the cell equations increases throughput. Since the multipliers and adders are word computation elements, all bits of the input word must be accumulated before computation can begin. This is a serious drawback in some signal processing environments due to slow ADC characteristics. This is not so with SBNR PE's as proposed herein because computation begins when the first converted bit becomes available (and not the last converted bit).

11.0 SBNR Implementation

A mesh connected systolic array of SBNR cells is used to implement McWhirter's algorithm. A single SBNR cell is shown in Figure 14. The design of this cell is based on the GAPP systolic array design. It consists of an appropriate set of registers which act as input to an intermediate SBNR ALU and a final SBNR ALU.

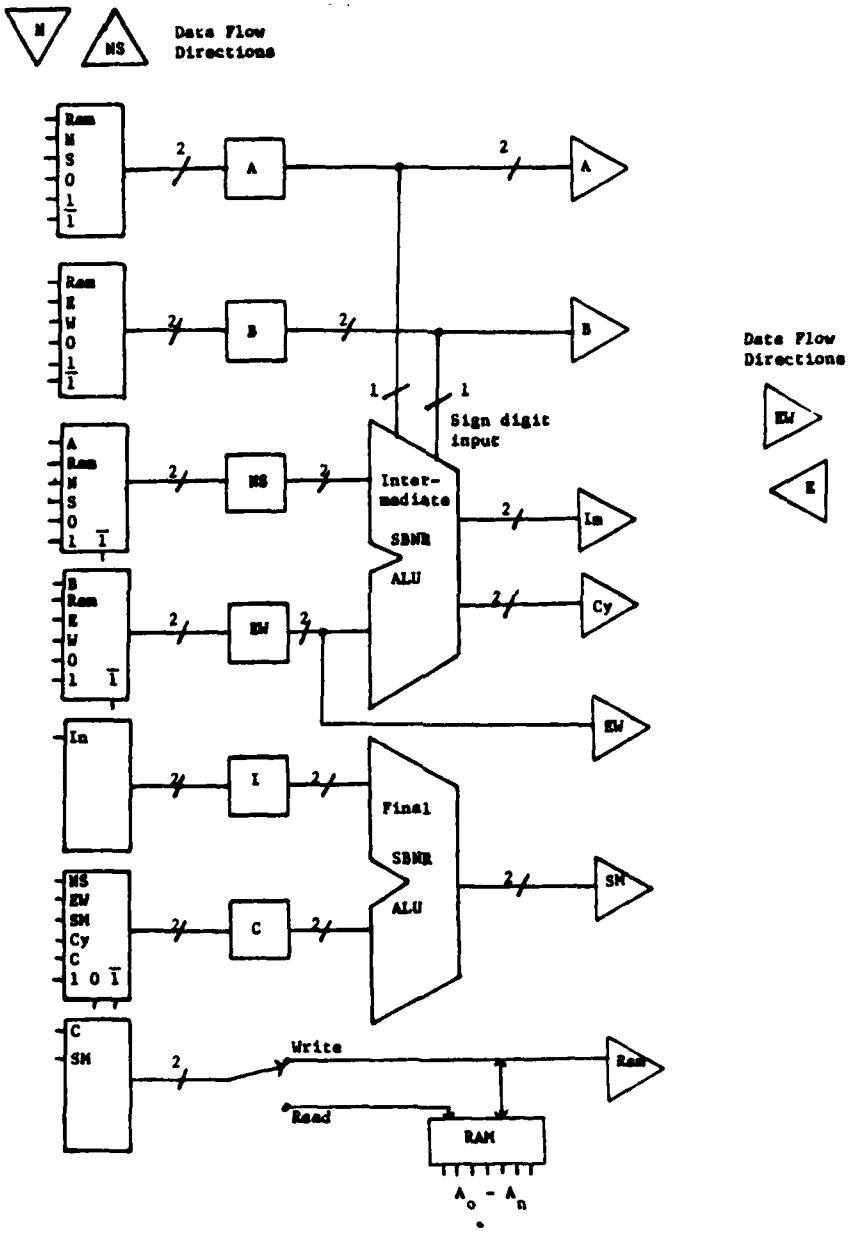


Figure 14 An SBNR Primitive Cell for a 2-D Mesh Connected Array

To perform a two operand computation using this SBNR element on the qth digit of the operands requires the following sequence of steps:

1. Load X_q, Y_q in the EW and NS registers.

2. Load X_{q-1}, Y_{q-1} in the A and B registers.

3. The intermediate ALU produces a partial result of X_q o Y_q at the Im output. "o" represents an arithmetic computation.

4. Latch the Im output in the I register.

5. Latch X_{q-1}, Y_{q-1} (currently in A,B) in the EW and NS registers.

6. Load X_{q-2}, Y_{q-2} in the A and B registers.

7. The intermediate ALU produces a partial result of X_q o Y_q at the In output. A carry from this operation occurs at Cy.

8. The carry Cy from partial computation of X_{q-1} o Y_{q-1} is latched in the C register.

9. Final computation of X_q o Y_q occurs at the final ALU. The result appears at SM.

From this algorithm, it can be seen that production of the first result digit requires that the q, q-1, and q-2 operand digits must be accumulated before the qth result digit is produced. Furthermore, if there exist N dependent equations such that equation N-1 depends upon results from equation N then 2N+1 operand digits from operands of the Nth equation are required before the first result digit is produced from equation 1 (the lowest equation on the data dependency tree). Latency is directly dependent upon these results and is expected to be 2N+1, given a system of N dependent equations.

Two techniques suggest themselves in the solution of a set of equations. The first is to use each processor to compute results for the entire equation set. In this case, bit serial processing is delayed by N+3 digits. Additionally, a certain amount of independent operation control is necessary for each processor. This is because a geometrically regular problem, when considering each operand as a word, becomes irregular when operands are processed bit serially. The second technique is to perform the computation for each equation in an equation set at a separate processor. This approach suffers from geometric irregularity problems and often results in violations of local communication restraints in systolic arrays.

Geometric irregularity in bit serial processing might be overcome using temporal skews on the bit data much as matrix operands are skewed to solve data dependencies in array manipulations. It is also possible to use mask bits in a systolic array which help to define the type of operation a cell performs or to signal that data is available. These masks could be propagated along with the bit data and, in effect, implement an MIMD systolic array which operates appropriately on each data type and availability. A more complex and possibly faster SBNR implementation of McWhirter's Givens rotation can be developed by

considering maximally parallel computation of the equations at each processor type (boundary, internal, and final cells). By allowing simultaneous computation of equations that are not data dependent, three architectures can be designed using SBNR primitive cells which implement each of McWhirter's systolic cells. Figures 15, 16, and 17 show the architectures for the boundary, internal, and final cell architecture. In a detailed circuit implementation, appropriate delays must be placed between each systolic array cell as well as between each SBNR processor internal to the systolic cells.

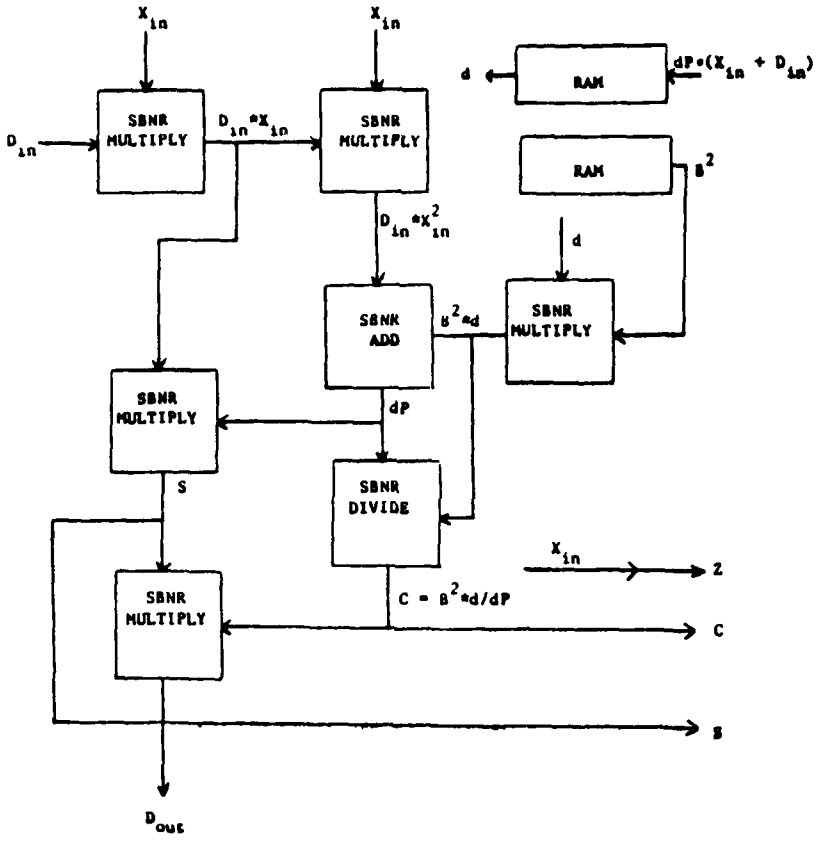


Figure 15 SBNR Boundary Cell Architecture

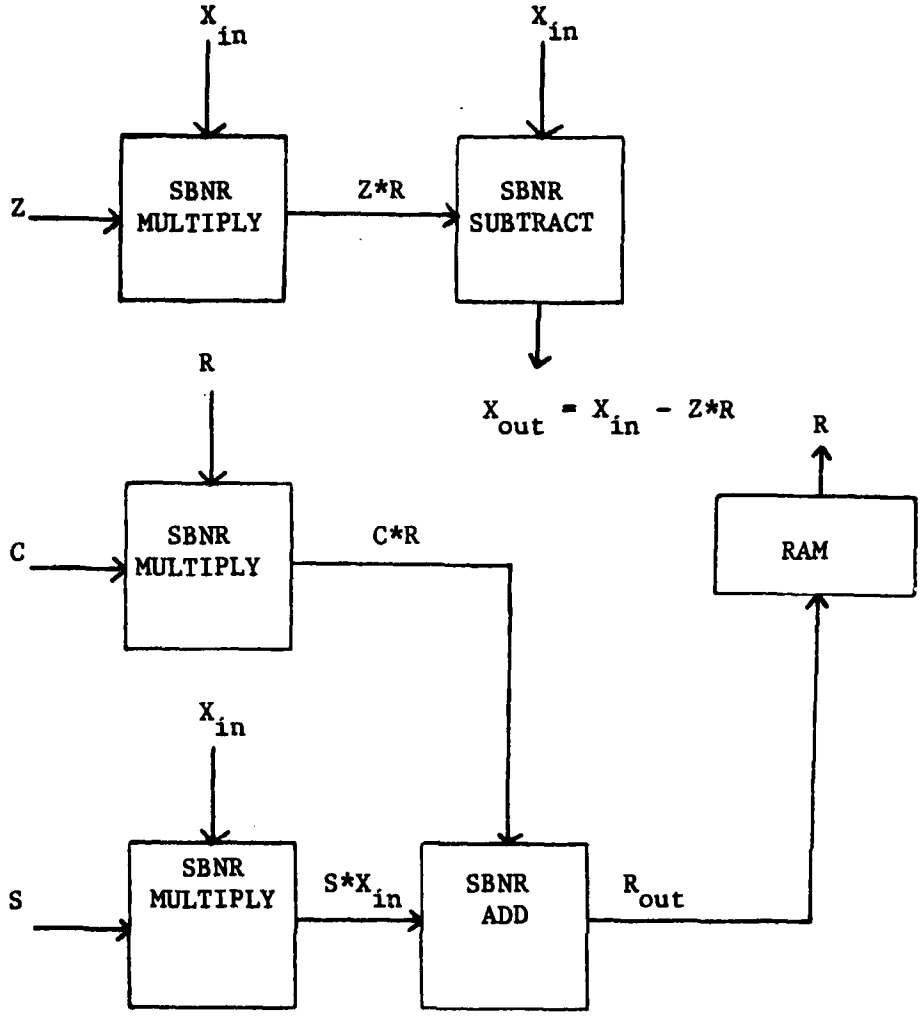


Figure 16 SBNR Internal Cell Architecture

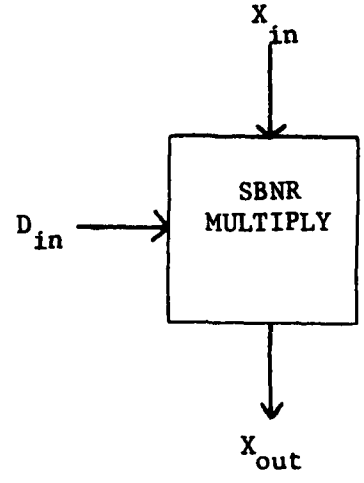


Figure 17 SBNR Final Cell Architecture

It is possible to derive the minimum execution speed and latency for this particular SBNR implementation of McWhirter's systolic array by considering the data dependencies in the equations. The maximum data dependency path length for the boundary, internal, and final cells are 5, 2, and 10, respectively. Using the 2N+1 formula for latency, we can compute latencies and speed estimates for each cell.

The maximum boundary cell latency is 11 cycles. At the internal cell, the maximum latency is 5 cycles. At the final cell, the maximum latency is 3 cycles. If r is the number of rows and c is the number of columns, then the maximum latency to the first residual is:

$$L(c,r) = 11(c + r + 1) \tag{45}$$

The execution time for the entire Givens rotation is:

$$S(c,r,n) = (2r + c + 1)(11 + n - 1) \tag{46}$$

where n is the bit length of the operands. Notice that the execution time is linearly dependent ($O(n)$) on the bit length of the operands where the GAPP array execution time is quadratically dependent ($O(n^2)$). This is a result of the reduction of multiplication complexity in SBNR arithmetic.

12.0 Distributed Arithmetic Implementation

The goal in distributed arithmetic architectures is to reduce computation time by performing table look-up to produce partial computation results. For example, multiplication can be reduced to a table look-up of partial products followed by a series of shift and adds to obtain the final result. In an N bit by N bit multiply, it is possible to divide each operand into k segments. By combining each segment of one operand with every other segment of the other operand, an address in RAM of each partial product is formed. The partial products are looked up and, through a series of shifts and adds, accumulated to form the final product. Table 4 shows a comparison of a typical N bit multiply using conventional binary versus distributed arithmetic.

Table 4 N Bit By N Bit Multiply Comparison

Conventional Binary Operations	Distributed Arithmetic Operations
n shifts	k^2-1 shifts
n adds	k^2-1 adds
n^2 ands (bit-wise)	k^2 table look-ups

If RAM access speed is greater than the computation time for n^2 AND operations, then distributed arithmetic provides better performance than conventional arithmetic for $k < n$. There is a trade-off between table size and computation speed. The table size for any distributed arithmetic multiplier is $2^{2n/k}$ words. While computation time is directly proportional to k, the table is indirectly proportional to k (i.e., large k implies larger computation time but smaller table size).

An n bit distributed arithmetic computational element is shown in Figure 18. This computational element is a single bit ALU with the special feature that a table address register can be loaded and a partial product retrieved for further computation.

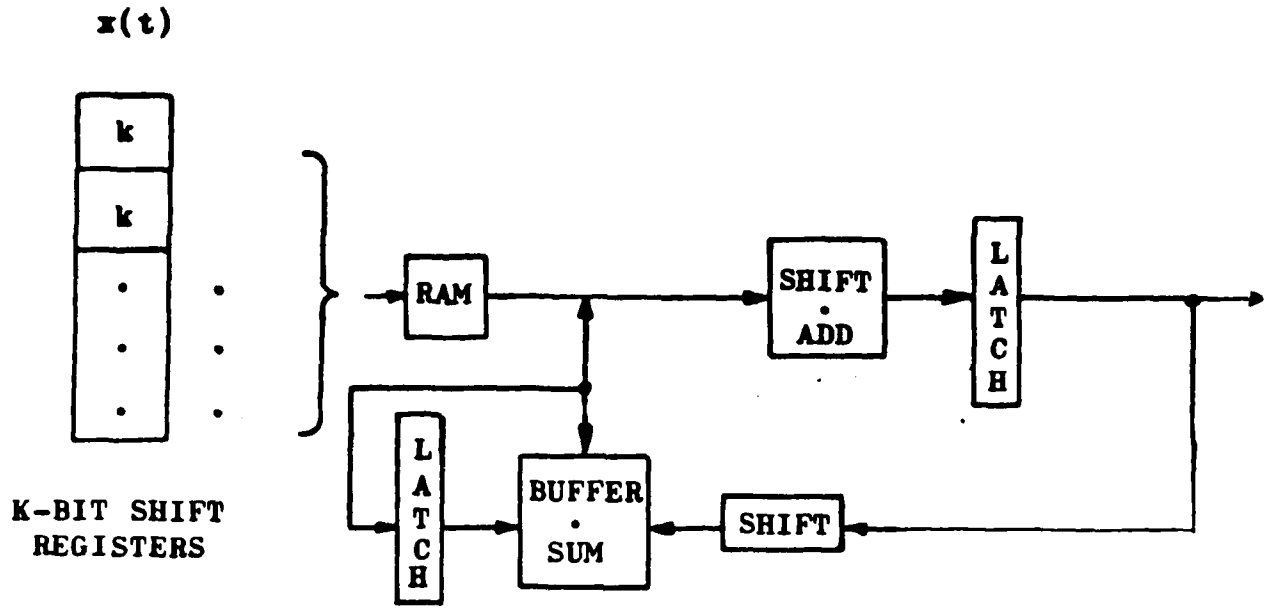


Figure 18 Distributed Arithmetic Primitive Element Architecture

This cell can be incorporated in a mesh-connected systolic array to perform the Givens rotation by McWhirter's algorithm. This cell can be assumed to operate like the bit-sequential cell of the GAPP array except that the multiplication is no longer $O(n^2)$ but $O(n)$. An estimate of the latency and execution time can be made by removing the n^2 term from the estimates made for GAPP. Thus the latency for a systolic distributed arithmetic implementation of McWhirter's algorithm is approximately

$$(c + r + 1)(224n + 156) \tag{47}$$

and the execution time is approximately

$$(2r + c + 1)(224n + 156) \tag{48}$$

A distributed arithmetic architecture might be constructed which would look up prestored quantized rotators rather than computing sine and cosine factors in the Givens rotation. This technique could save up to 5 multiplication/division operations in the boundary cell of McWhirter's systolic array. Whether this concept can be efficiently implemented remains an open development question.

13.0 Oversized Array Processing

So far we have considered signal matrices which are dimensioned no larger than the computational systolic array. Adaptive beamforming problems may arise where the signal matrix is larger in one or both dimensions than the processor array. We refer to such signal matrices as oversized matrices. We may think of partitioning an oversized matrix as taking a window over the matrix. The window is a submatrix of the oversized matrix. As the window is moved across the oversized matrix computations occur. As the window is moved the submatrix computations approach the desired computation for the entire array. It is important to consider the windowing of the matrices carefully because of data dependency requirements which may exist between computation results of one window of the matrix and the processing of a neighboring window. Multipass processing using previous results may be imperative to the solution of some data dependencies.

An example of data dependency which must be considered when processing matrices in a Givens rotation are the propagation of sine and cosine results across the columns of the matrix. Additionally, computation of sine and cosine at the edges of the current matrix require elements from preceeding rows. These dependencies must be carefully analyzed in order to deduce a working partitioning scheme for adaptive beamforming.

Improvements in multipass processing efficiency may be possible. Heller [32] discusses additive splitting techniques to decrease bandwidth, increase density, and use symmetry properties to improve processing efficiency. Additive splitting techniques may prove effective in the processing or large signal matrices for the Givens rotation. A thorough analysis is proposed for future work in systolic adaptive beamforming.

14.0 Error Analysis

Most of the implementations analyzed in this study are fixed-point implementations. Those implementations with short word lengths exacerbate round-off and truncation. In order to be absolutely certain that overflow and underflow do not aggravate computational results, an error analysis should be performed. This analysis is not within the scope of this current study.

In any event, an error analysis necessary for completeness must address the following issues. There are basically two different sources of error. The first is due to the effects of finite word length arithmetic. The second is due to quantization of the input samples.

If the signal-to-noise ratio (SNR) is relatively high, and the computational sequences seldom generate underflowed results, we know that finite word length arithmetic effects can be treated as Gaussian independent variables [33-36]. Hence, linear operations on such variables will only generate additive noise sources which obey the laws of homogeneity and superposition. Furthermore, such noises are zero mean, uncorrelated, and each variable has a variance of $2^{-2n}/3$ where n is the word length in bits chosen to represent the precomputed sum of partial

products. The output noise variance becomes a transformation of the computational noise variances. We speculate that a Jacobian transformation is satisfactory, thus minimizing error analysis. However, a study of the complete chain of sequences, identifying any multiplicative operations that multiply our linear operator assertions is necessary.

The quantization noise due to finite input representations (analog-to-digital conversions) is well-understood for conventional 2's complement arithmetic. For SBNR, further analysis is needed. In the case of 2's complement numbers, let x be the quantized (rounded) version of X to B bits, including sign bit and x^i be the ith bit. The quantization error is $e' = x - X$ with $-2^{-B} < e' < 2^{-B}$. The quantization error, e', is an uncorrelated sequence with zero mean and a variance of $2^{-2B}/3$ [37].

In order to estimate the effect of rounding errors, our least-squares equations should be posed as an unweighted least-squares problem. Then if the error is analyzed for a sequence of orthogonal transformations (as they are to some extent in this study), the results on error analysis of orthogonal transformations in [37,38,39] can be directly applied. A classical technique of backward error analysis from [39] is possible in order to obtain the lim sup of noise errors. The bounds are determinable when we desire:

a. The condition number of the matrix under question $||X|| \, ||X^T||$ where X^T is the Moore-Penrose inverse of X.
b. The norm of the residual $||Xw + y||$.
c. The norm of the backward error in X.
d. The norm of the backward error in y.

Of course the two quantities are dependent on the signal space and not on the algorithm. It is the last two quantities that depend on the algorithm as well as the precision of the underlying arithmetic.

In short, if we bound c. and d. above, we can show that the scaled Givens rotation endorsed in this study and implemented in finite word length arithmetic produces an implemented algorithm that is as stable as any available. Hence, future studies should focus on the establishment of these two norms.

15.0 VLSI Design Tools

The cost of complex VLSI implementation of adaptive beamforming system necessitates software tools for design, simulation, and verification. For the purposes of designing VLSI for adaptive beamforming, tools which handle algorithm mapping to, and simulation of, systolic array architectures are useful. Such tools should:

a. Accept a concurrent algorithm specification.

b. Provide architecture descriptions of the algorithm at a number of hierarchical design levels [40]. This allows progressive verification and fine tuning from high-level abstraction to gate-level implementation.

c. Accept constraints at each design level.

d. Provide a common language for both design and constraint specification at every design level.

e. Incorporate a model of computation oriented for systolic architecture design.

f. The algorithm specification language should be high-level and executable [41,42].

g. Provide for simulation of the design at each hierarchical level.

Several design tools have been implemented but unfortunately, are not available commercially. Chen and Mead [43] present a framework for specification of non-linear systems with memory which spans design levels from high-level algorithm specification to transistor circuits. Melhem [44] discusses a simulator which solves a System of Causal Equations (SCE) specification of the algorithm.

VLSI implementations allow the programmer to take advantage of algorithm concurrency. Thus, it is necessary to use a notation or programming language which permits precise and workable concurrent algorithm specification.

Barr and Brentrup demonstrate the suitability of Ada as a high-level programming language for multiple processor applications [45]. Ada allows specification of signal processing problems in terms of high-level abstraction of data and control structures. Ada's multi-tasking constructs permit expression of parallel processing tasks where they arise.

Because of the acceptance of Ada and VHDL as military standard languages, we believe future work should include a study to determine the suitability of Ada for algorithm specification and VHDL for design specification at multiple hierarchical levels in a VLSI design tool. In Section 17 we discuss the suitability of Ada for expressions of adaptive beamforming algorithms.

16.0 Gate Counts

CMOS Gate estimations for VLSI implementations of the architectures presented in this paper have been made using industry conventions. The industry conventions follow in Table 5.

Table 5 Gate Count for Basic CMOS Functions

Basic Function	Gate Equivalence
Inverter	.5 or 1
1 to 5-input NAND	.5 per input
6 to 10-input NAND	.5 per input + 2
1 to 5-input NOR	.5 per input
6 to 10-input NOR	.5 per input + 2
1 to 4-input AND	.5 per input + 5
5 to 8-input AND	.5 per input + 2.5
1 to 4-input OR	.5 per input + .5
5 to 8-input OR	.5 per input + 2.5
Exclusive OR/NOR	2.5
AND-OR-INVERT	.5 per input
OR-AND-INVERT	.5 per input
2-to-1 mux	2

Table 6 shows the gate counts of the basic building blocks used in comparing our architectures.

Table 6 Basic Building Blocks of Systolic PE

w x n RAM	$\log_2 w + 6w + 32n$ gates
n bit register	$32n^2$ gates
SBNR partial adder	49 gates
SBNR final adder	15 gates
1 bit full adder/subtractor	16 gates
1 SBNR PE	$396 + \log_2 w + 6w$

17.0 Ada Based Grammar for Adaptive Beamform Applications

A significant problem in the area of simulation, design, and microcode generation from parallel algorithms for parallel hardware configurations is the low degree of portability of HLL code. Because no HLL parallel coding standards exist, most parallel languages have been designed with a degree of parallelism matched with a particular target machine [46]. These languages are not portable and unsuitable for expressing parallelism in a retargetable environment.

High speed processing architectures are a moving target as VLSI technology and concepts in processor organization continue to evolve. As a result, new language and translation systems must be revised continually to keep up with architectural changes. Significant time and effort is expended when an algorithm must be recoded to reflect changes in language structure.

A solution to this problem is to produce a HLL standard for parallelism expression. Given such a standard , code would be readily transported to new architectures. A standard would also support the development of retargetable code generation tools, design tools, and simulators.

Though we do not attempt to propose a standard in this section we are analyzing an Ada based grammar for its suitability in expressing Adaptive Beamform algorithms for microcode generation, design, and simulation over a spectrum of architectures. Ada has been demonstrated to be appropriate for expression of parallel processing in signal processing tasks [44]. Further demonstration is necessary to determine if Ada is appropriate for expression of dataflow algorithms for design, simulation, and retargetable code generation tasks.

Adaptive beamform algorithms are highly parallel and typically dataflow oriented. A HLL language to express these algorithms should then possess the following properties:

a. Possess a natural, high degree of parallelism expression.
b. Have both strong object (data) and strong action (processing and flow) expressiveness.
c. Though explicit machine dependence should be avoided, the language should facilitate a close interaction between the software and hardware structures.
d. Both global synchronization (clocked) and dataflow synchronization should be supported.

17.1 Degree of Parallelism

During the generation of microcode, VLSI design, or simulator structures from a formal algorithm expression, the degree of parallelism should decrease or remain equivalent from the formal specification to the generated code or design system. This is because increasing the degree of parallelism during a translation process requires analysis [47]. This has proven to be a costly and frequently ineffective approach. Because VLSI is a moving target, and simulation or design constraints are

unpredictable, the degree of parallelism available in the target system may be unknown at the formal specification time of the algorithm. Hence, it is important to select a language which exhibits a high degree of parallelism and to express any algorithm in as highly a parallel a manner as is possible in the given language.

Ada provides two features which aid in the expression of parallelism. The first is the **task** concept which allows the definition of a process which may be concurrently executed with other tasks. A **task** may be used to define the function of a single processor in a network of multiple processors. Multiple **task** declarations are possible and could be mapped to unique locations on a computational network. The task concept has been largely derived from the principles of communicating sequential processes. A technique of message passing is available for synchronization of **task** processes. This feature of parallelism control will be discussed further below.

The second feature is the **package** concept. A **package** separates logical program or algorithm units from other such units. This is particularly useful in separating machine resource groups and tasks. Hence, the **package** can separate the resources of each processor in a multiprocessor network or to further isolate multiple network configurations. The **package** allows definition of strict communication constraints between other **packages**. This accurately corresponds to restricted communication constraints found on many hardware configurations such as the local communication restriction of systolic arrays.

17.2 Object and Action Expression

Ada is both strongly object and action oriented. Object oriented programming involves concentration on the representation of the objects in the processing environment. Such objects might be the signals from and array of antenna, the object(s) which the signals are imaging, etc. Ada provides facilities for data abstraction and hiding necessary to perform object oriented programming. These features include:

a. Definition of enumeration types.
b. Definition of composite types.
c. Declaration of access pointers.
d. Declaration of private values.

Enumeration types allow the grouping of related objects into sets. Composite types allow the construction of records to logically contain object attributes. Access pointers permit dynamic creation, destruction, and management of objects. Private values permit limited access to values. This aids in protecting data from misuse. Adaptive beamforming can make use of all these object description features. Some examples are:

a. A series of desired signal frequencies can be coded as an enumeration type.
b. An antenna's attributes can be coded into a single composite record. Attributes might include size, frequency response, location, etc.
c. Access pointers can be used to maintain a dynamically

reconfigurable set of signal inputs from an array. Removal or addition of signals can then occur during run time give appropriate hardware capabilities.

d. Private values can be designed to limit access to gain or filter values to specified subprograms or users.

An additional object oriented feature is the ability to define data representation at the lowest levels. Ada allows definition of types which are sequences of bits, addresses, etc. Such low level data control is critical for defining attributes such as signal representation.

Action oriented programming is concerned with the manipulation of the objects described at the object programming level. Ada provides the following action oriented support:

a. Iterative looping
b. Recursive calls
c. Case and if decision making
d. Mathematical operators with the versatility and power of Fortran operators.
e. Multi-tasking with task communication and synchronization.

These features give Ada a high degree of flexibility for expressing algorithms.

17.3 Mapping Ada Software to Hardware Structures

Ada supports a close relationship between program structures and hardware structures because of the highly modular language structure. The presence of isolated blocks of code, such as tasks or packages with protected resources, shared resources, and distinct communications interconnections implies the structure of isolated hardware functional blocks with both protected and shared resources and hardware communication lines.

Ada also supports some low-level machine dependent features which help in mapping software data objects to hardware data objects. These are at the very lowest levels and include the mapping of record specifications, enumeration type specifications, length specifications and address specifications. These features may not be sufficient for assignment of tasks to specific processors in a network. This is a critical point if a computational network is executing an algorithm which requires a specific geometric layout of tasks. The Givens rotation for adaptive beamforming is a good example of this type of problem. The network consists of three different task types arranged in a triangular array (Figure 11). These tasks types are required to reside at specific processor locations.

The communication links established by the coded algorithm, along with the constraint that the array obey local communication constraints, define the geometry of the Given's rotation array. A compiler with a sufficient degree of intelligence could derive the geometric layout of such processors given the task communication links and the local communication constraint. The level of intelligence may be beyond current

AI concepts or computationally expensive. It may be of some value to allow a grammar extension which permits processor labeling much as the current grammar permits address labeling. Tasks could then be assigned to a particular processor at the low-end of program construction. The development question which remains open is whether a machine dependent specification in the high-level program is more or less desirable that analysis expense in the compiler.

17.4 Process Synchronization

For computational networks it is desirable to permit two kinds of synchronization. Dataflow synchronization is established by the arrival of data. A computational node does not operate until all of its inputs are present. This type of synchronization is directly supported by the communicating sequential processes nature of task communication. A task may execute an "accept" or a series of "accept" instructions which force a task to wait until appropriate data are received before it continues processing.

Global task synchronization is not directly supported but can be simulated using a clock broadcasting function which sends a "clock pulse" to every task. Compile time translation of global synchronization may prove to be difficult.

17.5 Ada Grammar Highlights

Ada appears to offer most of the features necessary to express adaptive beamformer algorithms for parallel processing environments including systolic arrays, hypercubes, and other computational network structures. Mapping of tasks to specific processor locations may prove to require a language extension. This is a recommended topic for future development. Global synchronization specifications are unnatural and require further study to determine what if any existing or extension features could be used to express globally synchronized processor/task operation.

18.0 Crucial Adaptive Beamforming Algorithms

In this section we present two systolic algorithms crucial to adaptive beamforming systems. These algorithms are the Fourier Transform and Convolution. We have chosen to use a matrix formulation of the Discrete Fourier Transformation in the development of the systolic Fourier Transformation algorithm. The systolic Convolution algorithm discussed is essentially the same as the algorithm presented in [48]. The Fourier Transform algorithm is used to transform time domain sensor data to frequency domain data for processing by a frequency domain beamformer. The Convolution algorithm is used in FIR filter implementation. The FIR filter decreases the sampling rate required in time domain beamforming.

18.1 Discrete Fourier Transform

A straightforward technique of obtaining a one-dimensional Discrete Fourier Transform (DFT) on a systolic array architecture is to produce a matrix multiply formulation of the transform and then use the matrix multiply systolic array presented in Section 6. The matrix multiply formulation of the DFT is well known but Eqs. (49-55) are included for completeness.

The DFT is given by the relation:

$$X(n) = \sum_{k=0}^{N-1} x(k)\, e^{-j2\pi nk/N} \qquad (49)$$

where $n = 0,1,\ldots,N-1$

Let

$$W = e^{-j2\pi/N}$$

and Equation (49) becomes

$$X(n) = \sum_{k=0}^{N-1} x(k)\, W^{nk} \qquad (50)$$

where $n = 0,1,\ldots,N-1$

Expanding Equation (50) yields the set of equations:

$$X(0) = x(0)W^0 + x(1)W^0 + x(2)W^0 + \ldots + x(N-1)W^0 \qquad (51)$$

$$X(1) = x(0)W^0 + x(1)W^1 + x(2)W^2 + \ldots + x(N-1)W^{(N-1)} \qquad (52)$$

$$X(2) = x(0)W^0 + x(1)W^2 + x(2)W^4 + \ldots + x(N-1)W^{2(N-1)} \qquad (53)$$

$$X(3) = x(0)W^0 + x(1)W^3 + x(2)W^6 + \ldots + x(N-1)W^{3(N-1)} \qquad (54)$$

$$\vdots$$

$$X(N-1) = x(0)W^0 + x(1)W^{(N-1)} + x(2)W^{2(N-1)} + \ldots + x(N-1)W^{(N-1)(N-1)}$$

The matrix representation of this system is:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ \vdots \\ X(N-1) \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & \cdots & W^0 \\ W^0 & W^1 & W^2 & \cdots & W^{(N-1)} \\ W^0 & W^2 & W^4 & \cdots & W^{2(N-1)} \\ W^0 & W^3 & W^6 & \cdots & W^{3(N-1)} \\ \vdots & & & & \\ W^0 & W^{N-1} & W^{2(N-1)} & \cdots & W^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ \vdots \\ X(N-1) \end{bmatrix}$$

or

$$\bar{X}(n) = \bar{W}^{nk}\bar{x}(k) \tag{55}$$

where the overbar indicates matrix quantities.

This is a matrix matrix or matrix vector multiply which can be solved using a complex valued variation of the systolic matrix multiply given in Figure 6.

A two-dimensional DFT is a linear operation given by:

$$X(u,v) = \sum_{j=0}^{N-1} \sum_{k=0}^{N-1} X(j,k)e^{-j2\pi(uj+vk)/N} \tag{56}$$

for all $u = 0,1,\ldots,N-1$
and $v = 0,1,\ldots,N-1$

This can be reduced to a series of one-dimensional DFT's taken over u and v (rows and columns) of the signal matrix.

18.2 Convolution

The convolution operation is given by:

$$c_i = \sum_{j=0}^{i} a_j\, b_{i-j} \tag{57}$$

for all $i = 0,1,\ldots,N$.

A systolic convolution algorithm can be formulated to perform this computation. The systolic method chosen for the convolution algorithm involves piping the **a** and **b** factors from left to right into the systolic array. The **b** terms are moved twice as fast as the **a** terms. Whenever a b_i "catches up" with element a_o it is copied at the cell where the rendezvous occurs. This copied **b** is used in latter computations. The copied b_i and the current a_j at every cell are multiplied and summed with the current c_i. c_i is shifted from right to left at the same rate that a_j is being

shifted from left to right. The convolution algorithm is presented below:

```
Do
{
     shift as right one cell
     shift bs right two cells
     at the leading edge {cell where a0 occurs} copy b to
     memory location copied b
     compute    c = c + a * copied b  at every cell
     shift cs one cell left
}
Until 2N-1 cs "fall off" the left edge
```

Figure 19 illustrates the convolution algorithm.

Initial Conditions

$$\begin{array}{l} a_2\ 0\ a_1\ 0\ a_0 \\ b_4\ b_3\ b_2\ b_1\ b_0\ 0\ 0 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ c_0 & 0 & c_1 & 0 & c_2 \end{bmatrix}$$

First Iteration

$$\begin{array}{l} a_2\ 0\ a_1\ 0 \\ b_4\ b_3\ b_2\ b_1 \end{array} \begin{bmatrix} a_0 & 0 & 0 & 0 & 0 \\ b_0 & 0 & 0 & 0 & 0 \\ b_0 & 0 & 0 & 0 & 0 \\ c_0 & 0 & c_1 & 0 & c_2 \end{bmatrix}$$

{move as 1 cell}
{move bs 2 cells}
{copy b}
{$c_o = a_o b_o + c_o$}

Second Iteration

$$\begin{array}{l} a_2\ 0\ a_1 \\ \quad b_4\ b_3 \\ \\ \quad\quad c_0 \end{array} \begin{bmatrix} 0 & a_0 & 0 & 0 & 0 \\ b_2 & b_1 & b_0 & 0 & 0 \\ b_0 & b_1 & 0 & 0 & 0 \\ 0 & c_1 & 0 & c_2 & 0 \end{bmatrix}$$

{move as 1 cell}
{move bs 2 cells}
{copy b}
{move c 1 cell;
compute:
$c_1 = c_1 + a_0 b_1$;
}

Third Iteration

$$\begin{array}{l} a_2\ 0 \\ \\ \\ c_0\ 0 \end{array} \begin{bmatrix} a_1 & 0 & a_0 & 0 & 0 \\ b_4 & b_3 & b_2 & b_1 & b_0 \\ b_0 & b_1 & b_2 & 0 & 0 \\ c_1 & 0 & c_2 & 0 & c_3 \end{bmatrix}$$

{move as 1 cell}
{move bs 2 cells}
{copy b}
{move c 1 cell;
compute:
$c_1 = c_1 + a_1 b_0$;
$c_2 = c_2 + a_0 b_2$;
}

Computation continues until all c_i for i = 0 to 2N-1 are computed.

Figure 19 Systolic Convolution Example

19.0 Adaptive Beamformer and Tracker System

Feintuch, et. al. [49], investigated an adaptive tracking system which employed the LMS algorithm to minimize the error between two beams of a split array (Figures 20a and 20b). The weights generated are analyzed to determine the peak of the weights. The peak of the weights roughly correspond to the delay between the phase centers of the two beams. The phase or time-delay is then used to provide a bearing estimate.

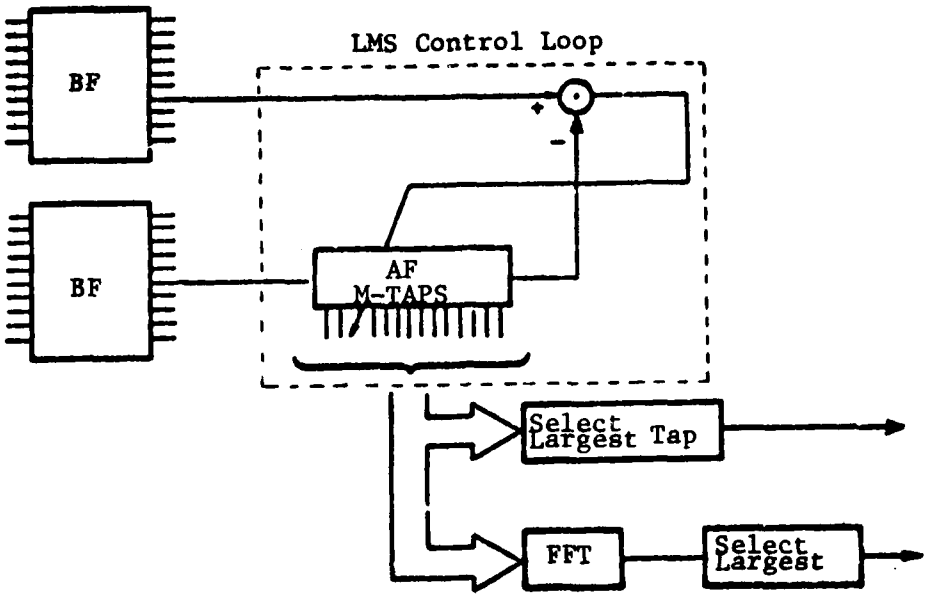


Figure 20a LMS Time Domain Adaptive Tracker

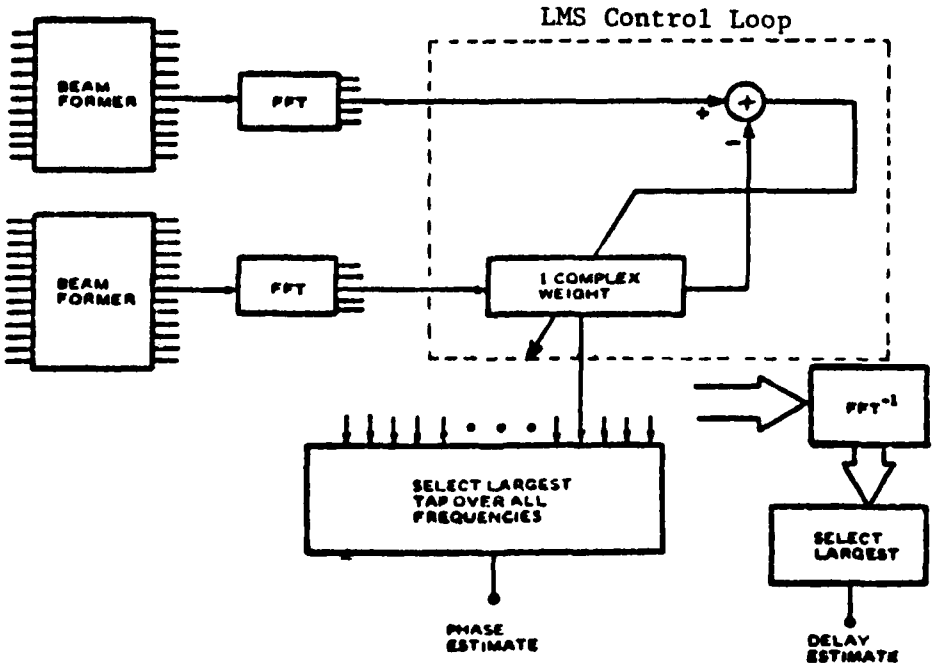


Figure 20b LMS Frequency Domain Adaptive Tracker

In this study, we want to consider a least-square implementation of the adaptive tracker. We then wish to construct a completely systolic adaptive beamformer/tracker from the systolic Givens rotation, DFT, and backsubstitution architectures studied in this report. The Feintuch study provides a suitable starting point for incorporating adaptability. Simply stated, use the peaks between weights to electronically steer the beam to force nulls at jammer angles.

19.1 Least-Squares Adaptive Tracker

A time-domain least-squares adaptive tracking system can be developed by considering the block diagram of Figure 20a where we replace the LMS control loop with an LS algorithm block. This configuration is shown in Figure 21a. Two inputs are required for the adaptive tracker:

$y(n)$ - the beam from one half the array at time nT where T is the sampling rate and n is the current time index.

$\underline{X}(n) = [x[nT], x[(n-1)T], \ldots, x[(n-M+1)T]]^T$ - the last M samples from the other half of the array beginning with the current time index n to time index $(n-M+1)$.

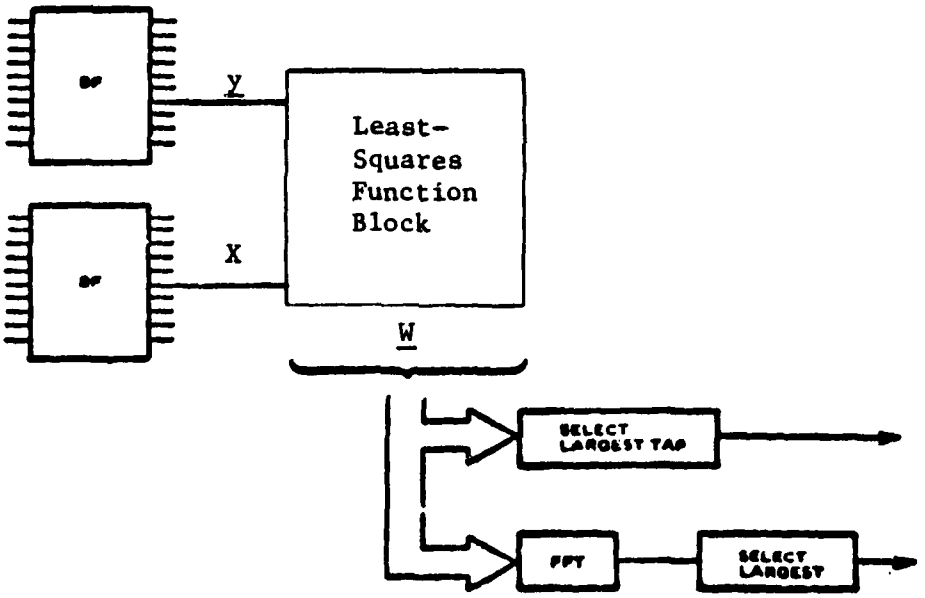


Figure 21a Least-squares Time Domain Adaptive Tracker

Multiple time "snapshots" of these samples are collected to form the LS algorithm inputs $\underline{y}(n)$ and $\underline{X}(n)$. The LS algorithm computes the weight vector $\underline{W}(n)$ which minimizes the least-squares norm:

$$E(n) = ||\underline{e}(n)|| = ||\underline{X}(n)\underline{W}(n)+\underline{y}(n)||$$

The largest tap of the weight vector is then found and provides the delay or phase bearing estimate.

In the frequency domain solution, shown in Figure 21b, M inputs from the time domain beam undergo a Fourier transform to produce the M frequency components. Multiple time "snapshots" of one half the array are taken to produce a matrix of frequency components for the LS algorithm. The largest tap over the frequency weights is selected and the phase provides a bearing estimate which is used to steer the beam.

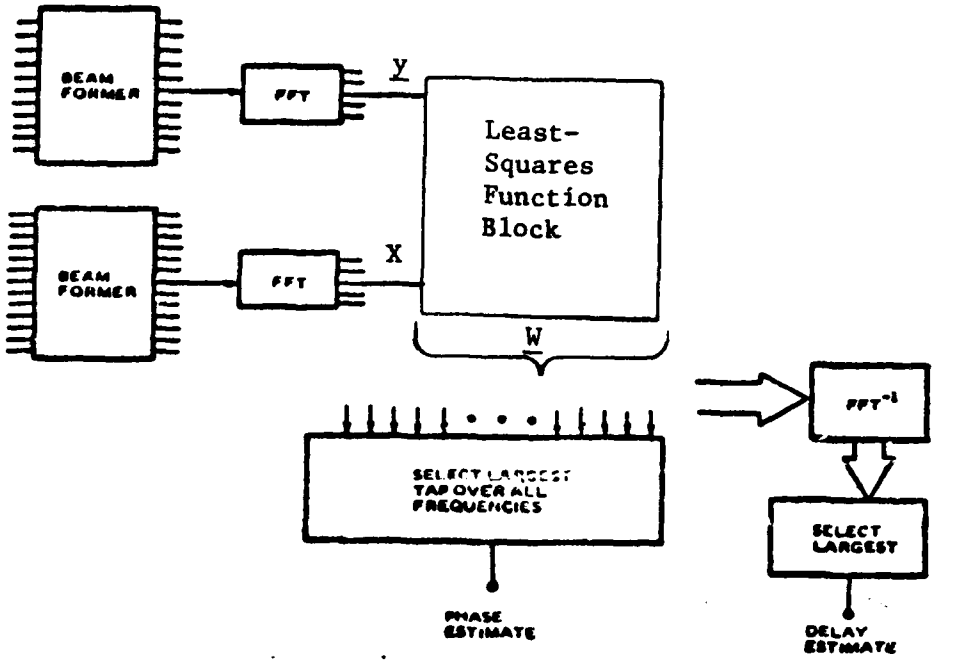


Figure 21b Least-squares Frequency Domain Adaptive Tracker

There are several advantages to performing the adaptive tracking in the frequency domain. Mean and variance of the weights can be predicted, the time domain weight variance can then be deduced given proper choice of LS parameters. This cannot be done in the time domain. Computational savings is realized in the frequency domain because of the fact that the M frequency components are complex conjugate symmetric. Hence, only M/2 points need to be included as LS inputs.

19.2 Systolic Adaptive Beamformer and Tracking System

Figures 22a and 22b show a complete frequency-domain adaptive beamformer and tracking system. The computational extensive components of the system are systolic array modules.

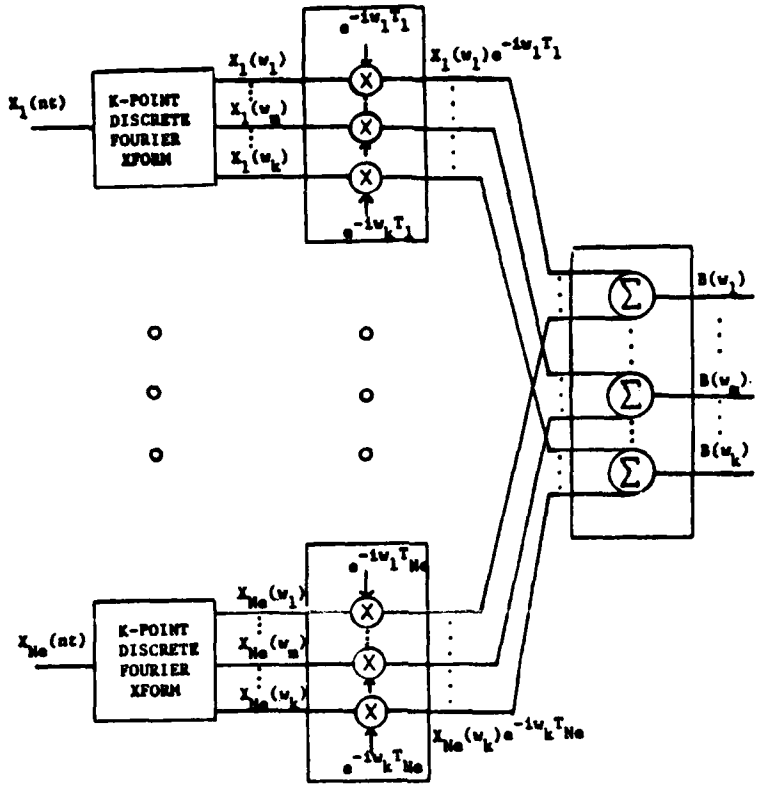


Figure 22a Frequency Domain Adaptive Beaformer

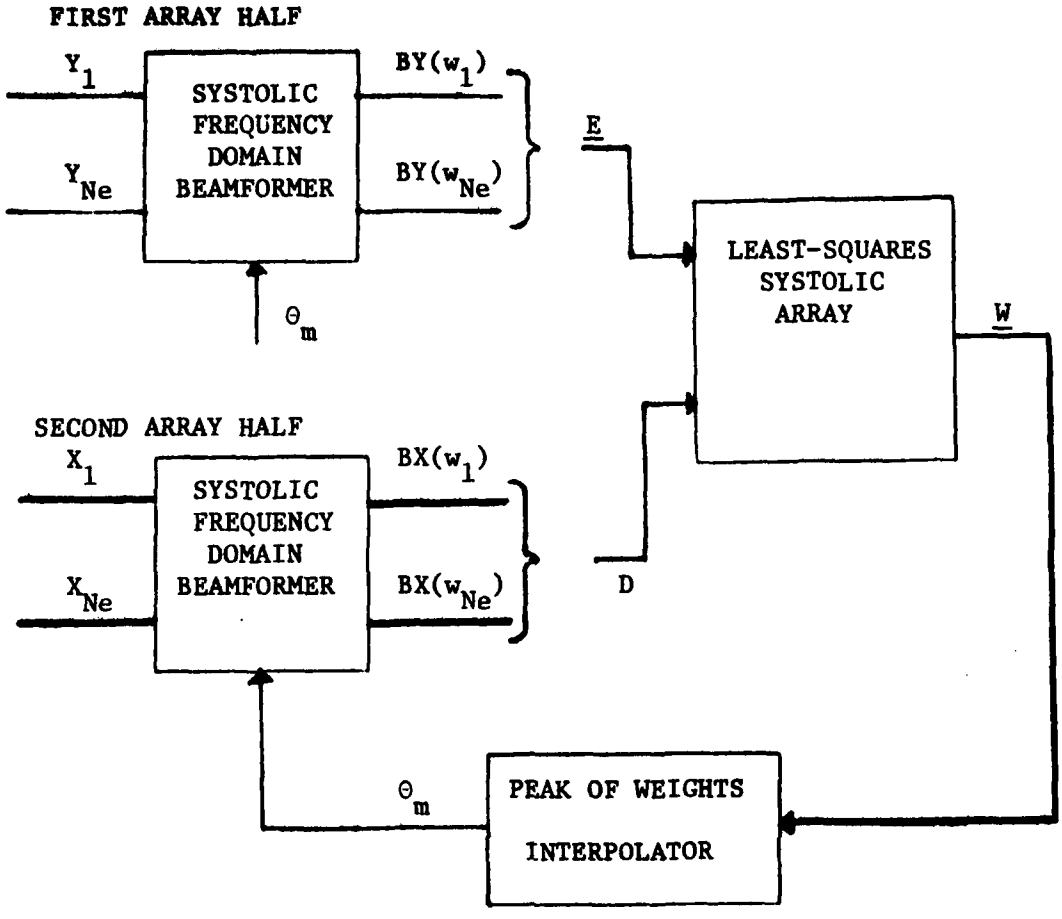


Figure 22b Frequency Domain Adaptive Beamformer and Tracker

The K-point DFT modules of the system perform a Fourier transform of the time domain input data. This systolic array may be the DFT systolic array discussed in Section 18.1 or some other Fourier transform such as the systolic FFT presented by Willey, et. al., in [50]. A system consideration at this point is the Fourier transform throughput. If too slow, the sampling rate may be decreased below the Nyquist rate which is the minimum rate required for the frequency domain solution. The Nyquist rate must be maintained to prevent aliasing. The matrix multiply solution presented in Section 18 may prove too slow, require extensive buffering, or require multiple systolic arrays working in a frame buffered configuration. Willey's FFT may prove more efficient because of the highly pipelined nature of computation (i.e., a signal frame is piped through the FFT and multiple signal frames may be processed in earlier pipeline layers). Further study is required to determine the most efficient solution.

The phase shift multiply is driven by the phase estimate from the adaptive tracker. Each frequency component from the Fourier transform is multiplied by the term

$e^{-iw_m T_m}$

where T_M is a function of the steering angle θ_m. A multiplier array is the component which operates at this function block. A conventional distributed arithmetic engine or SBNR distributed arithmetic engine may be ideal for this array since W_m is only position dependent and θ_m is the only variable and non-position dependent quantity in the computation of T_m for a fixed sensor array. Hence, an ultra fast table look-up of $e^{-iw_m T_m}$ can occur based solely on the steering angle θ_m. Each look-up table must be customized for each frequency component.

An adder array is used to form the frequency bins of the beam. Since the beam is already in the frequency domain after the summing operation, the bins can be fed directly to the LS algorithm. The LS block consists of a Givens rotation systolic array (Section 8) and a backsubstitution array to compute the weights.

The peak of the weight vector can be found using a quadratic interpolator. The interpolator performs a quadratic fit to the largest weight element and the two adjacent weights in the frequency domain. Feintuch, et. al., present a quadratic interpolator in [49]. From the located peak, a bearing estimate is extracted from the phase of the complex valued weight. This estimate is fed back to the phase shifter.

The adaptive beamformer and tracking system depicted in Figures 22a and 22b can be described in an algorithmic format. This algorithm is useful for understanding the processing flow and the required data inputs. This algorithm could be easily ported to microcode or simulator code given a finer resolution of the details of each phase.

ADAPTIVE BEAMFORMER AND TRACKER ALGORITHM

Variable Dictionary:

R - increments through an indefinite duration of time steps (as long as the tracker operates).

n - indexes time "snapshots" for forming the second half of the array signal matrix.

TS - the number of time snapshots chosen.

s - indexes the sensors.

K - number of points in the DFT transform.

X_s - a vector of K sensor points for sensor s in the second array.

Y_s - a vector of K sensor points for sensor s in the first array.

QY_s - the frequency domain phase shifted data from sensor s in the first array half.

B - a vector of beams (one for each K frequency component).

D - a matrix (TS x K) of TS time "snapshots" of K beam frequency components from the second array half - used as the matrix input to the LS algorithm.

$\underline{E}$ - a vector of K beam frequency components from time TS - used as the vector input to the LS algorithm.

I - intersensor spacing.

t - indexes time steps for collection of K data points.

$S_{1s}(q)$ - sensor input from first array half, sensor s, at a given time q.

$S_{2s}(q)$ - sensor input from second array half, sensor s, at a given time q.

FX_s - the Fourier transform of the K points of input data from a sensor s of the second array half.

FY_s - the Fourier transform of the K points of input data from a sensor s of the first array half.

QX_s - the frequency domain phase shifted data from sensor s in the second array half.

θ_m - the current beam steering angle.

C - propagation velocity of the incoming signal.

Ne - number of sensors in an array half.

Variable Dictionary (continued):

w_m - a frequency component.

$\underline{W}$ - the vector of adaptive weights generated in the LS algorithm.

DFT - discrete Fourier transform function.

LS - least-squares algorithm function.

PEAK - an interpolator function for finding the phase of the peak of the adaptive weight.

Algorithm:

```
/*The following algorithm performs a tracking sequence of indefinite
duration*/

initially θ_m is user selected

for R = 1 to Duration of tracking
{
   /*collect a series of time "snapshots" for second half of the array*/
   for n = 1 to TS do sequentially
   {
      /*at all sensors collect K samples*/
      for s = 1 to Ne do parallel
      {
         for t = 1 to K do sequentially
         {
              X_s(R+n+t) = S_2s(R+n+t)
         }
      }
      /*for all sensors of the second array half perform a DFT of the
      data*/
      for s = 1 to Ne do parallel
      {
              FX_s = DFT(X_s)
      }

   /*for all sensors of the second array half perform a parallel phase
   shift realizing the beam steering for each frequency component*/
   for s = 1 to Ne do parallel
   {
      for m = 1 to K do parallel
      {
      /*perform phase shift*/

              QX_s(w_m) = FX_s(w_m) e^(iw_mTm)
              where Tm = s(d/c) cos θ_m
      }
   }

   /*form the frequency components of the beam*/
   /*initialize B(w_m) first*/
   for m = 1 to K do parallel
   {
              B(w_m) = 0
   {

   /*now sum to form the beam components*/
   for m = 1 to K do parallel
   {
      for s = 1 to Ne do sequentially
      {
              B(w_m) = B(w_m) + QX_s(w_m)
      }
```

```
}
/*after the beam components are formed, collect the "snapshot" for
this n in a matrix*/
          Dn = B
{

/*during the formation of "snapshot" TS collect a single "snapshot" of
beam components from the first array half*/
/*for each sensor collect K samples*/
for s = 1 to Ne do parallel
{
   for t = 1 to K do sequentially
   {
          Y_s(R+TS+t) = S_1s(R+TS+t)
   }
}

/*for all sensors perform a DFT*/
for s = 1 to Ne do parallel
{
          FY_s = DFT(Y_s)
}

/*for all sensors of the array perform a phase shift to realize the
beam steering for each frequency component*/
for s = 1 to Ne do parallel
{
   for m = 1 to K do parallel
   {
      /*perform the phase shift*/
          QY_s(w_m) = FY_s(w_m) e^(jw_m Tm)
          where Tm = s(d/c) cos θ_m
   }
}

/*form the frequency components of the beam*/
/*initialize B(w_m) first*/
for m = 1 to K do parallel
{
          B(w_m) = 0
}

/*now sum to form the beam components*/
for m = 1 to K do parallel
{
   for s = 1 to Ne do sequentially
   {
          B(w_m) = B(w_m) + QY_s(w_m)
   }
}

/*collect this beam as a vector*/
E = B
```

```
    /*now perform the LS algorithm generating the weight vector W as a
    product*/
    W = LS(D,E)

    /*find the peak of the weights and use the phase as a new estimate for
    the tracking beam*/
    O_m = PEAK(W)
}
/*end of algorithm*/
```

20.0 Resolution Enhancement of Digital Beamformers

In many cases, it is desirable to improve beam pattern without extending the array length. Characteristics of the beam pattern and beamformer SNR depend on array length and number of sensors. For a fixed sensor separation, D, more sensors mean larger arrays. If physically limited (e.g., by aircraft width or length), one approach is to use synthetic aperture radar (SAR). However, instabilities in straight path trajectories degrade SAR performance. Modern spectral estimation (MEM, MLM, etc.) often called "superresolution" beamforming, in contrast to conventional beamforming, have merit [51-55]. Actually these techniques are more of a detection and spectral estimation than of beamforming. However, Fan, et. al., [55] have been able to use some signal extrapolation techniques to generate explicit values outside of the given region (so as to extend the received signal of a fixed array beyond its actual physical length.

These considerations affect the architectural choices of an adaptive beamformer. Tantamount among them is the interface circuitry complexity between the analog and digital circuitry. For simplest circuitry, the SBNR arithmetic engine interfaces via ADC and DAC with minimal complexity. Hence, digital beamformers which were ruled out because they necessitated too many ADC's and DAC's are now cost effective again. Specifically, delay-sum and partial-sum approaches are now competitive rivaling the cost effectiveness of interpolation beamformers. No increase in cost effectiveness has been realized for frequency domain approaches incorporating SBNR realizations favorable for ADC/DAC's. However, frequency domain approaches which, by the way, eliminate the need for high input sampling rate, typically invoke DFT's. Our systolic array implementations take advantage of the SIMD nature of the multiple sensor streams of input data to make microprogrammable control simple and fast. Now system flexibility so nicely captured in the Raytheon beamformer also exist in our systolic array realizations.

21.0 A Microprogrammable Digital Beamformer (MDB)

Raytheon has developed an MDB which incorporates the partial-sum concept as shown in Figure 23. The major feature is an incrementing counter under programmable microcode which addresses partial-sums dynamically. This research tool is useful as a generic testbed for accommodating on-line changes in beam and sensor number, shading (weighting), beam steering, etc. Some of its features can be easily captured in systolic array implementations, especially the flexible microprogrammability. An adaptive beamformer architecture can then utilize microprogram control not only for optimal beam steering, but also for changing beam, sensor, and jamming environments. Furthermore, other algorithms besides the LS (such as conjugate gradient within the least mean square algorithm) can be invoked with minimal hardware redesign.

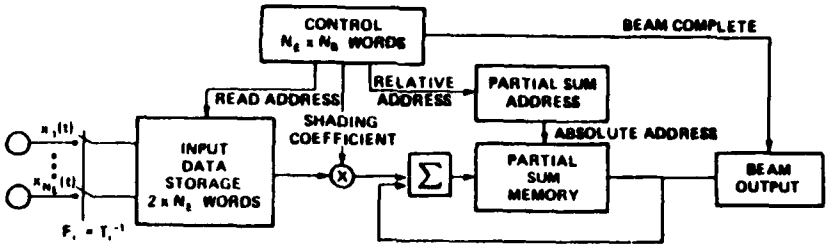


Figure 23 Discrete-Time, Partial-Sum Beamformer

22.0 Comparative Analysis

Table 7 compares several least-squares computational techniques. The normal equations, Householder, Golub factorizations, standard Givens rotation, fast Givens rotation, scaled Givens rotation, and Gram-Schmidt methods are considered. Either the normal equations or the Householder Golub techniques require global communications. Additionally, these two techniques are sensitive to ill-conditioned matrices. Hence, the normal equations or the Householder Golub method are not amenable to systolic implementation. The Gram-Schmidt method, included for completeness, is not recursive and, therefore, is not considered for systolic implementation.

The remaining methods are based on the Givens rotation triangular decomposition. The standard Givens rotation requires pivoting as well as square-root computation. This slows the computation on systolic arrays. The square-root free Givens rotation eliminates the square-root computation but still requires pivoting. The scaled Givens rotation eliminates both the square-root computation and pivoting. Additionally, the scaled Givens rotation operates on matrix bands. It is not necessary to perform any computation on bands that contain only null elements. A computational savings is realized if the data matrix is in banded form. Note that the square-root free and scaled Givens rotations require half as many multiplies as the standard Givens rotation. The scaled Givens rotation only requires 1 division operation as opposed to 2 in the square-root free rotation. Apparently the scaled Givens rotation is superior to the other methods studied both in terms of computation speed and systolic implementation complexity.

Table 7 Weighted Least Squares Computational Methods

	Normal Eqn's	House-Holder Golub	Triangular Decompositions: Standard Givens Rotations	Triangular Decompositions: Fast Givens Rotations	Triangular Decompositions: Scaled Givens Rotations	Gram-Schmidt
Systolic Amenable	non-nearest neighbor data paths	requires global comm	yes, but is slow and processor complex recursive separate back-substitution systolic array.	If factored $\sqrt{\ }$ free operation, nearest-neighbor pivoting increases data flow complexity	No pivots and $\sqrt{\ }$ free, nearest neighbor comm.	not recursive
Additions/ Subtractions						
Mult./Stage			N	N/2	N/2	
Div./Stage			2		1	
Shifts/Im	Scal. Compl	Scal. Compl	Complex	Complex	2	
Latency				r+c+1		
Stable	Sensitive to Matrix Ill-condition Number			Yes	Equiv. to standard Givens	well cond.
Pivoting			2x1 Vector	2x1 Vector	None	
Fading Signal Capacity (weighted)	Complex	Complex	Complex	Simple	Simple	
Row Removal	Complex	Complex	Complex	Complex	Simple	
Idle Processors			N/2	N/2	N/2	
Computation Time				2r+c+1	$3m+3(q-1)+z+1$	$^{1}O(m+z)$
Number of Processors				c(r+1)/2	q(w+z)	$O(w^2+zw)$

Table 7 Notation:

Assumptions:

A	m x n matrix, rank n to machine precision, m > n
S	m x 1 vector
V	n x 1 vector
P	number of subdiagonals
q	number of superdiagonals
w	number of bands, bands dense and narrow w = p+q+1
z	number of righthand side vectors
c	columns of rectilinear matrix
r	rows of rectilinear matrix
processing elements	planar orthogonally connected systolic array performing elementary operations {+,*,/}, nearest neighbor mesh

Notes:
1. With q(w+z) processors, computation time is 2m + 3(q-1)+z+1 for scaled Givens rotations.

A comparison of systolic primitive elements is presented in Table 8. Five architectures are examined.

a. Conventional binary bit sequential cell (GAPP)(Section 10.0)
b. Conventional binary (complex cell)(Section 10.0)
c. Distributed arithmetic (Section 12.0)
d. Signed binary number representation (complex cell)(Section 11.0)
e. Signed binary number representation (mesh-connected PE's)(Section 11.0)

The architectures were studied on the basis of a square-root free Givens rotation implementation to obtain speed and latency estimates. The conventional binary (complex cell) and SBNR (complex cell) are both algorithm dedicated architectures. As a result, they have irregular structures and are not VLSI amenable. The conventional binary (complex cell) architecture has O(1) speed and latency. The SBNR (complex cell) and the SBNR (mesh-connected PE), which, by the way, is VLSI amenable, have O(n) and O(1) speed and latency, respectively. The distributed arithmetic architecture exhibits O(n) speed and latency.

The conventional binary (complex cell) is superior in terms of speed and latency only. The distributed arithmetic and SBNR (mesh-connected PE) architectures have excellent speed and latency and both are VLSI amenable. Distributed arithmetic bandwidth is smaller than SBNR (mesh-connected PE) bandwidth, however, SBNR (mesh-connected PE) has a superior latency. For adaptive beamformer implementation, both architectures are excellent. Careful weighing of engineering trade-offs must be made to establish any clear superiority of the distributed arithmetic or SBNR (mesh-connected PE's) architectures.

Table 8 Comparison of Systolic Array Architectures

Complexity and Performance Cell Type

	Conventional Binary Bit Seq. Cell (GAPP)	Conventional Binary (1 Adders)[1] (3 Multipliers)	Distributed Arithmetic
Speed	$(2r+c+1)(83n^2+224n+156)$	$3(2r+c+1)$	$(2r+c+1)(224n+156)$
Latency	$(r+c+1)(83n^2+224n+1)$	$3(r+c+1)$	$(r+c+1)(224n+1)$
Cell Complexity	Simple	Complex	Simple
I/O Bandwidth	c	cn	c
VLSI Amenable	Yes	structure irregular	Yes
Algorithm Dedicated	No	Yes	No
Gate Counts	--	--	--

1. For Boundary Cell. Internal cell requires 2 Adders, 2 Multipliers. Final cell requires 1 Multiplier.

	SBNR (Complex cell)	SBNR (mesh-connected PE's)
Speed	$(2r+c+1)(20+n)$	$O(n)$
Latency	$20(r+c+1)$	$O(1)$
Cell Complexity	Complex (multiple SBNR PE's)	Simple
I/O Bandwidth	2c	2c
VLSI Amenable	structure irregular	Yes
Algorithm Dedicated	Yes	No
Gate Count	$rc(2768 + 7\log_2 w + 42w + 64n)$	$rc(187 + \log_2 w + 6w)$

Table 8 Notation:

r - rows of rectilinear matrix
c - columns of rectilinear matrix
n - word length

Table 9 compares and contrasts some efficient beamforming techniques. The techniques are compared on the basis of their spectral properties, sampling rate required, the number and type of systolic modules required for systolic implementation, predictability of mean and variance of weights in adaptive systems, type of beam steering, and a qualitative measure of ADC complexity and data storage requirements.

The systolic array modules listed in Table 9 require a brief explanation. A time delay pipeline is a linear array. As sensor samples are piped down the array, a time delay of i is realized by using the sample stored at the ith array element as input to the rest of the beamformer. An adder array is a linear array of adders used to perform multiple addition operations in parallel. A convolver is a convolution systolic array such as the one presented in Section 18.2. A complex sampling filter is a systolic array implementation of a Hilbert transform, time delay, or lowpass filter. The Fourier transform array is a systolic array such as the one developed in Section 18.1. A multiplier array is a linear array of multipliers for performing parallel multiplications. Finally a narrowband filter is a systolic array implementing a narrowband filter.

Time domain beamforming methods including delay-sum, partial-sum, and shifted sideband tend to require a high sampling rate, high ADC complexity, and large memory requirements unless augmented by interpolation or interpolation and complex sampling. In the frequency domain the mean and variance of the weights can be predicted. From these estimations jammer bearing and signal strength can be deduced. The time domain methods, especially delay-sum or partial-sum, without interpolation, require fewer systolic array modules than the frequency domain techniques. This is because frequency domain techniques require a Fourier transform to transform the time domain data to the frequency domain. Apparently, frequency domain techniques are superior for adaptive beamforming and tracking systems.

Table 9a

Beamforming Method	Spectral Bands	Sampling Rate	Autonomous Systolic Modules
Delay-Sum	Lowpass Bandpass	<5Ny[1]	Ne Time Delay Pipelines
Partial-Sum	Lowpass Bandpass	<5Ny	1 Adder Array
Interpolation	Lowpass	Ny	[Ne Convolvers + S][1] [1 Convolver + S][2]
Interpolation With Complex Sampling	Bandpass	Ny	1 Complex Sampling Filter[3] + I
Shifted Side Band	Bandpass	>5Ny	Complex (TBD)
Discrete Fourier Transform	Lowpass Bandpass	Ny	Ne+1 Fourier Transform (or Inverse) + 1 Multiplier Array + 1 Adder Array
Phase-Shift	Narrowband	Ny	1 Narrowband Filter + 1 Complex Sampling Filter + 1 Multiply Array + 1 Adder Array

Table 9b

Beamforming Method	Predictable Mean and Variance of Complex Weights[4]	Beam Steering Technique	Analog to Digital Converters[5]	Data Storage[6]
Delay-Sum	No	Time Delay	H	H
Partial-Sum	No	Time Delay	H	L-M
Interpolation	No	Time Delay	L	L-M
Interpolation With Complex Sampling	No	Time Delay	L	L
Shifted Side Band	No	Time Delay	L-M	L-M
Discrete Fourier Transform	Yes	Phase-Shift	L	H
Phase-Shift	Yes	Phase-Shift	L	L

Table 9a and Table 9b Notation:

Ny - Nyquist rate
Ne - Number of sensors
S - Number of systolic components from the backend beamformer
I - Number of systolic components in the specific interpolation technique chosen

Notes:
1 Pre-interpolation
2 Post-interpolation
3 Systolic implementation of Hilbert transform, time delay, or lowpass Filter - sampling technique dependent
4 Time domain techniques do not allow prediction of mean and variance of adaptive weights if adaptive filters are applied.
5 ADC and DAC complexity might be reduced in SBNR architecture implementations.
6 Systolic arrays tend to employ a large number of processing cells with memory at each cell. High data-storage generally implies that the technique is amenable to systolic implementation.

23.0 Conclusions

Table 6 suggests that the scaled Givens rotation algorithm appears to be superior to all others for systolic array implementation. This is evident by its square root-and pivot-free nature. This conclusion is only tentative because the control unit microprograms and circuitry must be considered. Furthermore, PE utilization needs to be studied. At present, only half of the PE's of a rectilinear array are active.

Table 7 indicates that SBNR architectures are superior to other architectures when considering both speed and overall complexity. Some of the qualities which make SBNR superior are:

1. Limited carry propagation during computations
2. O(n) multiply
3. Left to right or right to left bit-wise computations
4. Simple adder/multiplier

As yet, there is no commercially available VLSI implementing SBNR computation elements. It appears that at this time, GAPP is a current solution in terms of availability and density of computation elements per chip.

Table 9 suggests that frequency domain beamforming techniques may be superior to time domain techniques. Some advantages of frequency domain methods are:

a. Low sampling rate
b. Predictable and variance of complex weights
c. Low ADC complexity

Further study is required if clear superiority of either time domain techniques or frequency domain techniques are to be established.

23.1 System Considerations

An optimum method for selecting an implementation and, subsequently, an adaptive beamforming architecture is dependent on several related factors, each of which constitutes an engineering trade-off. System specifications often include the number of sensors, number of beams formed simultaneously, beam steering resolution, frequency band(s) of interest, postbeamforming processing, dynamic range, and fault tolerance. In this study we have attempted to consider system implementations for a few adaptive beamforming applications, those in which the number of beams and sensors are fixed and beam steering resolution is not critical. Although fault tolerance can be easily sustained in signed digit arithmetic, we have not performed an in-depth study of its properties. Rather we made use of the innate redundancy of the number system to force reduction in hardware. In this way, the reliability of the processing hardware increased.

Discrete time beamformers utilize considerably more hardware. A/D conversion, input data storage and beamformer computational circuitry are sensitive to the beamforming methodology. For example, the delay-sum

technique needs relatively large amounts of memory. Interpolative beamformers need lower input sampling rates and, often, less A/D circuitry. The SBNR arithmetic engine in the systolic array needs relatively trivial ADC and DAC circuits. Hence, fault tolerance and speed advantages naturally accrue.

Much of this study has focused on frequency domain beamforming (FDB) for several reasons. Beam steering resolution does not depend on input sampling frequency. Therefore, sensor data can be sampled at lower rates than interpolation beamforming. Another FDB advantage is found in its inherent spatial aspect. Computationally efficient FFT's can be used to compute the multidimensional DFT over both space and time. For phase shift beamformers, another alternative FDB, the input sampling rate and input data requirements are reduced, rendering very efficient system designs for narrowband beamforming. It might be advantageous to study a combination of MEM and MLM methods with extrapolation because the main lobe width still has an inverse relationship with the total number of sensors and the ratio of sensor separation divided by wavelength (of signal of interest). (Because characteristics of beam pattern and beamformer SNR are dependent on sensor number and array length, Fan, et. al. [55] sought to combine conventional digital beamforming with signal extrapolation techniques.)

The major thrust of this study was to identify efficiently fast adaptive beamforming algorithms and supporting architectures that capture the theoretical algorithmic speeds.

23.2 Status of Accomplishments

In this Phase I period, the following tasks were completed. A study of architectural features supporting fast real-time, fault-tolerant adaptive beamforming was executed. A selection of engineering trade-offs was made in order to compare systolic arrays amenable to frequency domain beamforming, primarily. However, all identified architectures work equally well for conventional time domain beamforming. A systolic array of signed binary number arithmetic engines serving as primitive elements (PE) appear to satisfy real-time requirements.

Microprograms testing matrix by matrix multiplication and other least squares algorithm intensive operations were written and tested in a small systolic array simulator. From this coding process, speed/memory parameters were established providing empirical comparative data on the NCR GAPP systolic array.

A VLSI model was derived and used to drive eventual system-wide designs that could identify relevant architectures. The VLSI model establishes complexity of design measures (minimal width wires, double layer metalization, ...) which are 1985 conservative estimates. Thus, the architecture floorplans and circuit routings can be improved as technology advances. This modeling task was undertaken in order to prescribe directions for chip layout circuits.

The major adaptivity task of beamforming is the computation of the LS algorithm. Hence, several algorithms were studied. A scaled Givens

rotation which is free of square roots appeared to be time and space optimal. Systolic array architectures were then identified which supported both least squares algorithms and FFT's (necessary for beamforming). Triple product convolution within frequency domain beamforming was also implemented as an FFT, pointwise multiplication, and an inverse FFT, all of which prove to be faster than straightforward convolution.

24.0 Recommendations

a. Build a systolic array and/or simulator to derive the timing specifications for an eventual VLSI device to achieve low power, microminiaturization for flyable beamformer circuits.

b. Generate the complete set of microprograms for a frequency domain adaptive beamformer in order to test out future algorithms. This tool set shall be ported to VAX machines under UNIX.

c. Build a firmware engineering tool base for systolic array designs that can perform complete engineering trade-offs of adaptive beamformer architectures.

d. Integrate the modern spectral estimation methods with conventional beamforming algorithms to improve extrapolative beam steering.

e. Prepare a breadboard adaptive beamformer configured with SBNR processing elements in systolic arrays (to support fault-tolerant designs) as the first design step to a brassboard and, eventually, a production lightweight beamformer, possibly serving as a front-end to the SDI processor. An objective is to demonstrate minimal intercell connections and maximal fault tolerance.

REFERENCES

[1] P.A. Businger and G.H. Golub, "Linear Least Squares Solutions by Householder Transformations," Num. Math., Vol. 7, pp. 269-78, 1965.

[2] L. Johnsson, "A Computational Array for the QR-Method, Proc. Conf. Advanced Research in VLSI," MIT, pp. 123-129, 1982.

[3] D.E. Heller and I.C.F. Ipsen, "Systolic Networks for Orthogonal Decompositions, SIAM J. Stat. Sci. Comp., Vol. 4, pp. 261-269, 1983.

[4] G.H. Golub and C.F. van Loan, Matrix Computations, The John Hopkins Press, Baltimore, MD, 1973.

[5] C.D. Thompson, "A Complexity Theory for VLSI," Ph.D. Dissertation, Carnegie-Mellon University, Computer Science Department, Pittsburgh, Pennsylvania, Aug., 1980.

[6] R.P. Brent and H.T. Kung, "The Area-Time Complexity of Binary Multiplication," Tech. Report CMU-CS-79-136, July, 1979.

[7] C. Mead, and L. Conway, Introduction to VLSI Systems, Reading, Massachusetts, Addison Wesley, 1980.

[8] J.V. McCanny, K.W. Wood, J.G. McWhirter, and C.J. Oliver, "The Relationship Between Word and Bit Level Systolic Arrays as Applied to Matrix x Matrix Multiplication," SPIE, Real Time Signal Processing VI, Vol. 495, pp. 114-120, 1983.

[9] A. Avizienis, "Signed-Digit Number[s] Representations for Fast Parallel Arithmetic," IRE Trans. on Electron. Computers, Vol. EC-10, pp. 389-400, Sept., 1961.

[10] D.E. Atkins, "Design of the Arithmetic Units of ILLIAC III: Use of Redundancy and Higher Radix Methods," IEEE Trans. on Computers, Vol. C-19, No. 8, pp. 720-733, Aug., 1970.

[11] C. Tung, "Signed-Digit Division Using Combinational Arithmetic Nets," IEEE Trans. on Computers, Vol. C-19, pp. 746-749, Aug., 1970.

[12] M.D. Ercegovac, "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer," IEEE Trans. on Computers, Vol. C-26, pp. 667-680, July, 1977.

[13] J.E. Robertson, "A New Class of Digital Division Methods," IRE Trans. Electron. Computers, Vol. EC-7, pp. 218-222, Sept., 1958.

[14] F.A. Rohatsch, "A Study of Transformations Applicable to the Development of Limited Carry-Borrow Propagation Adders," Dept. of Computer Science, University of Illinois, Urbana, Rept. 226, 1967.

[15] A. Avizienis, "On a Flexible Implementation of Digital Computer Arithmetic," Information Processing, C.M. Poppleweld Editor, Amsterdam, North Holland, 1963.

[16] A. Avizienis, "Binary-Compatible Signed-Digit Arithmetic," 1964 Fall Joint Computer Conference, AFIPS Proceedings, Vol. 26, pp. 663-672, 1964.

[17] A. Avizienis and C. Tung, "A Universal Arithmetic Building Element (ABE) and Design Methods for Arithmetic Processors," IEEE Trans. on Computers, Vol. C-19, pp. 733-745, Aug., 1970.

[18] J.E. Robertson, "Parallel Digit Arithmetic Unit Utilizing a Signed-Digit Format," U.S. Patent #3,462,589, 1969.

[19] N. Takagi, H. Yasuura, and S. Yakima, "A VLSI-Oriented High-Speed Multiplier Using a Redundant Binary Addition Tree," Systems-Computers-Controls, Vol. 14, pp. 19-28, 1983.

[20] C.Y.F. Chow, "A Variable Precision Processor Module," Ph.D. Dissertation, University of Illinois, Computer Science Dept., Urbana, 1980.

[21] J.E. Robertson, "Design of the Combinational Logic for Radix 16

Digit Slice for a Variable Precision Processor Module," Proc. 1983 IEEE International Conference on Computer Design, Port Chester, N.Y., Oct., 1983.
[22] C. Caraiscos and B. Liu, "A Round-Off Error Analysis of the LMS Adaptive
Algorithm," ICASSP 83, pp. 29-32, 1983.
[23] M. Andrews, "Comparative Implementations of the LMS Algorithm," submitted to Int. Journal of Computers and Electrical Engineering, 1985.
[24] A. Morgul, P.M. Grant, and C.F.N. Cowan, "Wide-Band Hybrid Analog/Digital Frequency Domain Adaptive Filter," IEEE Trans. on Acoustics, Speech, and Signal Processing, Vol. ASSP-32, pp. 762-769, Aug., 1984.
[25] I.R. Mactaggart and M.A. Jack, "A Single Chip Radix-2 FFT Butterfly Architecture Using Parallel Data Distributed Arithmetic," IEEE Journal of Solid-State Circuits, Vol. SC-19, pp. 368-373, June, 1984.
[26] K. Hwang and Y.H. Cheng, "VLSI Computing Structures For Large Scale Linear Systems of Equations, " Int. Conf. on Parallel Processing, pp. 217-227, Aug., 1980.
[27] A. Corry and K. Patel, "Architecture of a CMOS Correlator," IEEE Int. Symposium on Circuits and Systems, May, 1983.
[28] J.V. McCanny and J.G. McWhirter, "A Bit Level Systolic Array for Matrix x Vector Multiplication," to be published in IEE Part G, Electronic Circuits and Systems.
[29] J.G. McWhirter, "Recursive Least-Squares Minimization Using A Systolic Array," SPIE, pp. 106-112, August 1983.
[30] Kung, H.T. and Leiserson, C.E., "Systolic Arrays (for VLSI), Space Matrix Proceedings", pp.256-282, SIAM, Philadelphia, PA, 1978.
[31] J.L. Barlow and I.C.F. Ipsen, "Givens Rotation for the Solution of Linear Least Squares Problems on Systolic Arrays," Penn. State Univ., Univ. Park, PA, 1985.
[32] D. Heller, "Partitioning Big Matrices for Small Arrays," VLSI and Modern Signal Processing, pp. 185-199, Prentice Hall, Englewood Cliffs, N.J., 1985
[33] Claassen, T.A.L.M., W.F.G. Mecklenbrauker and J.B.H. Peck, "Some Considerations on the Implementation of Digital Systems for Signal Processing," Phillips Res. Reports 30, 1975, pp. 73-84.
[34] M. Andrews and R. Fitch, "Finite Word Length Arithmetic Computational Error Effects on the LMS Adaptive Weights", 1977, IEEE Int'l Conf., ASSP, Hartford, Connecticut.
[35] M. Andrews and D. Merchant, "C-Noise in Recursive Algorithms," 22nd Int'l Symposium SPIE, Aug., 1978, San Diego, CA.
[36] K.S. Kawarai, "Canonical Realization and Round-Off Noise Analysis for Combinational Filters," Elect. and Comm. in Japan, Vol., 65-A, No. 8, 1982, pp.19-27.
[37] Gentleman, W.M., "Least-Squares Computations by Givens Rotation Without Square Roots," J. Inst. Math. Appl., 12 (1973), pp. 329-336.
[38] ----------, "Error Analysis of QR Decompositions by Givens Transformations," Lin. Alg. Appl., 10 (1975), pp. 189-197.
[39] Wilkinson, J.H., "The Algebraic Eigenvalue Problem," Oxford University Press, London, 1965.
[40] M.C. Chen and C.A. Mead, "Concurrent Algorithms as Space-Time Recursion Equations," VLSI and Modern Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., p. 224, 1985.
[41] Ibid p. 227.

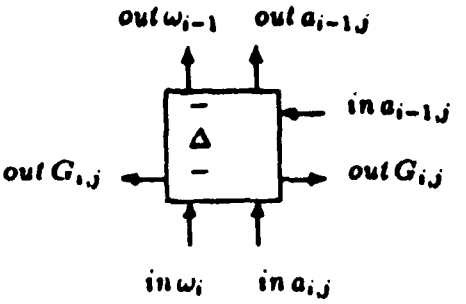
[42] A.B. Cremers and T.N. Hibbard, "Executable Specification of Concurrent Algorithms in Terms of Applicative Data Space Notation," VLSI and Modern Signal Processing, Prentice-Hall, Englewood Cliffs, NJ, pp. 200-201, 1985.
[43] M.C. Chen and C.A. Mead, "Concurrent Algorithms as Space-Time Recursion Equations," VLSI and Modern Signal Processing, Prentice-Hall, Englewood Cliffs, N.J., pp. 224-240, 1985.
[44] R. Melhem, "A Language For the Simulation of Systolic Architectures," Purdue University, Dept. of Computer Science, Private Memorandum.
[45] P.C. Barr and R.J. Brentrup, "Ada as a Signal-Processing Language," SPIE Vol. 241, Real-Time Signal Processing III, 1980, pp. 205-212.
[46] R.H. Hockney and C.R. Jessehope, "Parallel Computers," Adam Hilger Ltd., Bristol, pp. 213-214, 1981.
[47] Ibid, p. 213.
[48] J.D. Ullman, "Computational Aspects of VLSI," Computer Science Press, Rockville, Maryland, 1984, pp. 175-188.
[49] P.L. Feintuch, F.A. Reed, N.J. Bershad, and C.M. Flynn, "Final Report on Phase 3 of the Adaptive Tracking System Study," DTIC Distribution FR81-11-70, 1980.
[50] T. Willey, R. Chapman, H. Yoho, T.S. Durrani, and D. Preis, "Systolic Implementation for Deconvolution, DFT, and FFT," IEE Proceedings, Vol. 132, Pt. F., No. 6, pp. 466-472, 1985.
[51] R.N. McDonough, "Applications of the Maximum-Likelihood Method and the Maximum-Entropy Method to Array Processing," Nonlinear Methods of Spectral Analysis, Vol. 34, S. Haykin, Ed., New York: Springer-Verlag, 1979, Chapter 6.
[52] T.E. Barnard, "Two Maximum Entropy Beamforming Algorithms for Equally Spaced Line Arrays," IEEE Trans. ASSP, Vol. ASSP-30, April 1982, pp. 175-189.
[53] W.F. Gabriel, "Spectral Analysis and Adaptive Array Superresolution Techniques," Proc. IEEE, Vol. 68, June 1980, pp. 654-666.
[54] O.N. Strand, "Multichannel Complex Maximum Entropy (Autoregressive) Spectral Analysis," IEEE Trans. Automat. Contr., Vol. AC-22, August 1977, pp. 634-640.
[55] H. Fan, E. El-Masry, and W.K. Jenkins, "Resolution Enhancement of Digital Beamformers," IEEE Trans. ASSP, Vol. ASSP-32, October 1984, pp. 1043-1051.

Appendix A
Scaled Givens Rotation Algorithm

Algorithm:

Initially, $\Delta_{j'} := \Delta_{j'}^{(0)} := w_{j'}^{(0)}$; $\Delta_{h,j'} := s_{h,j'}$; $1 \le j' \le q$, $1 \le h \le z$;
for $i' = 1 \ldots m+q-2$ *do sequentially,*
for $j' = 1 \ldots \min\{i', q\}$ *do in parallel,*
if $q + i' - 2(j'-1) \le m$ *then*
$i := q + i' - 2(j'-1)$; $j := i' - (j'-1)$;
$\rho_j := \Delta_{j'} (a_{i-1,j})^2 + w_i (a_{i,j})^2$,
Compute α *and* β
where $\zeta_j := 1/\rho_j$, $\xi_j := \Delta_{j'} w_i \zeta_j$,
$G_{i,j} := \begin{pmatrix} 2^{\alpha}\Delta_{j'} a_{i-1,j} & 2^{\alpha} w_i a_{i,j}, \\ -2^{\beta} a_{i,j} & 2^{\beta} a_{i-1,j} \end{pmatrix}$,
$a_{i-1,j} := 2^{\alpha}\rho_j$, $a_{i,j} := 0$,
$w_{i-1} := 2^{-2\alpha}\zeta_j$, $\Delta_{j'} := 2^{-2\beta}\xi_j$, $\Delta_{j'}^{(0)} := w_i^{(0)}$,
for $l = 1 \ldots w-1$ *do sequentially by pipelining* $G_{i,j}$,
$\begin{pmatrix} a_{i-1,j+l} \\ a_{i,j+l} \end{pmatrix} := G_{i,j} \begin{pmatrix} a_{i-1,j+l} \\ a_{i,j+l} \end{pmatrix}$;
for $h = 1 \ldots z$ *do sequentially by pipelining* $G_{i,j}$,
$\begin{pmatrix} s_{h,i-1} \\ \Delta_{h,j'} \end{pmatrix} := G_{i,j} \begin{pmatrix} \Delta_{h,j'} \\ s_{h,i} \end{pmatrix}$.

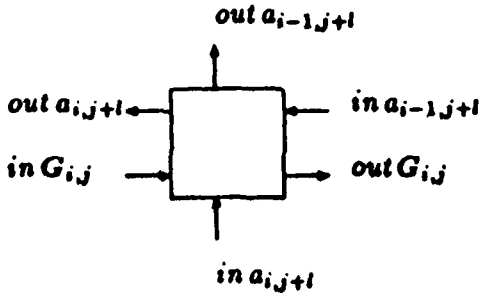
Scaled Givens rotation processors:



if in $a_{i,j} \neq$ '$-$' *then*
  $\omega_i :=$ in ω_i; $a_{i,j} :=$ in $a_{i,j}$;
  compute equations (5.1);
  out $\omega_{i-1} := \omega_{i-1}$; out $a_{i,j} := a_{i,j}$;
  out $G_{i,j} := G_{i,j}$.

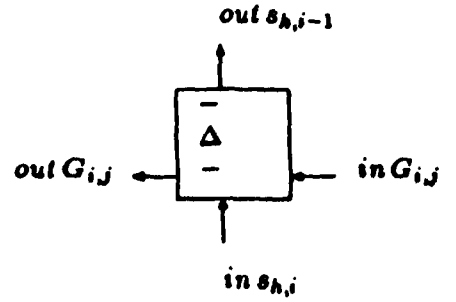
$\omega_{i-1} \equiv (w_{i-1}^{(0)}, w_{i-1})$, $\omega_i \equiv (w_i^{(0)}, w_i)$.
Δ Represents the Storage Cells $\Delta_{j'}$ and $\Delta_{j'}^{(0)}$.

(a) Matrix Processor $(j', 1)$ in Processor Vector j'.



if in $a_{i,j} \neq$ '$-$' *then*
  $a_{i-1,j} :=$ in $a_{i-1,j}$; $a_{i,j} :=$ in $a_{i,j}$;
  $G_{i,j} :=$ in $G_{i,j}$;
  compute equations (5.2);
  out $a_{i-1,j} := a_{i-1,j}$; out $a_{i,j} := a_{i,j}$;
  out $G_{i,j} := G_{i,j}$.

(b) Matrix Processors $(j', 2) \ldots (j', w)$ in Processor Vector j'.



if in $a_{i,j} \neq$ '$-$' *then*
  $s_{i,j} :=$ in $s_{i,j}$;
  $G_{i,j} :=$ in $G_{i,j}$
  compute equations (5.3);
  out $s_{i-1,j} := s_{i-1,j}$;
  out $G_{i,j} := G_{i,j}$.

Δ Represents the Storage Cells $\Delta_{h,j'}$ and $\Delta_{h,j'}^{(0)}$.

(c) Righthand Side Processors $(j', -z) \ldots (j', -1)$ in Processor Vector j'.

APPENDIX B

SQUARE-ROOT FREE GIVENS ROTATION GAL PROGRAM

```
/*************************************************************************
****
```

Procedure Name: Givens rotation

Purpose: This routine performs a least squares minimization using a Givens rotation strategy. The program does not do the actual back substitution required to solve for the weight vector but does calculate the least squares residual e. If the weight vector is required than another systolic array program can be used to solve for w from the upper triangular matrix **R** produced during the reduction of the input matrix. The algorithm used in this code is taken from McWhirter, "Recursive least-squares minimization using a systolic array."

Image Variables:

Flag 1, Flag 2 - one bit flags used during boolean tests

DI - delta in from Whirter's algorithm

DO - delta out from Whirter's algorithm

R - static matrix (upper triangular) produced during the reduction

XI - x element input

XO - x element output

CO - cosine terms

SI - sine terms

Z - z variable from Whirter's algorithm

d - d variable from Whirter's algorithm

dP - d prime variable from Whirter's algorithm

B - scale array referred to as beta in Whirter's equations

boundary_mask - defines boundary cells, zero active
internal_mask - defines internal cells, zero active
final_mask - defines final cell, zero active

Integer variables

```
     ; _  general purpose counter for loop construct

Execution cycles:  (2 rows + columns + 1) (83n^2 + 224n + 156)

       - where n is the bit length of the major operand XO, XI, etc.,
         and where rows and columns are the dimensions at the input
         matrix including the Y vector

Author:  Robert E. Boring
Revision:  1.0

*************************************************************************
****/


image  Flag 1:1, Flag 2:1;      /*boolean test flags */
image  DI:8, DO:8;              /* delta in, delta out*/
image  R:8;                     /* upper triangular matrix-static*/
image  XI:8, XO:8;              /* X in, X out */
image  CO:8, SI:8;              1* Sin and Cosine *1
image  Z:8;                     1* Z variable *1
image  d:8, dP:8;               1* d , d prime *1
image  B:8;                     1* Beta array (static input) *1
image  boundary_mask:1;         1* Static mask image - zero's
                                   define boundary PE's *1
image  internal_mask:1;         1* Static mask image - zero's
                                   define internal PE *1
image  final_mask:1;            1* Static mask image - zero's
                                   define final PE *1
int rows = 3, columns = 4;      1* x row, column dimensions *1

main ( )
{

     *1 before this code is executed the Beta array must be loaded and all
other ram locations cleared  *1.

     int i;

     for (i = 0; I < 2 rows + columns + 1; i ++)
     {

          1*  move the DO to DI from east must input a 1 to the
          top boundary cell at all iterations *1
               move _ e (DO,DI);

          1*  move X array one element south *1
               move _ s (XO,XI);

          1*  move cosine and sine one element east *1
               move _ e (c,1);
               move _ e (s,s);

          1*  move Z factor one element east *1
```

```
     move _ e (z,z);

1*  perform boundary cell operations *1
     calc_boundary (XI, DI, C, S, Z, DO, dP, d, B);

1*  perform internal cell computation *1
     calc_internal (XI,XO, DI, DO, R, C, S, Z, d, dP,
B);

1*  perform final cell computation *1
     calc_final (DI, XI, XO);

1*  move DO one element south to prepare for next
iteration *1
     move_s (DO, DO);

1*  repeat loop until all calculations are performed
*1

}

}  *1 END *1
```

```
/*************************************************************
*****************
```

Procedure Names: move_s, move_n, move_e, move_w

Purpose: These procedures move an image one cell south (move_s), north (move_n), east (move_e), or west (move_w). The input image **in** is moved into the output image **out**. This routine assumes that the input and output images are the same size.

Image Variables:

in - the image to be moved
out- the image to which the contents of **in** are moved.

Integer Variables:

i - counter for incrementing through each bit of the image.

Execution cycles: 3n - where **n** is the number of bits in the input image.

Author: Robert E. Boring
Revision: 1.0

```
*************************************************************
```

```
**********/

move_s (in, out)
image in, out;

{
   int i;

   for (i = 0; i < size(in); i++)
   {

          in:i    ns:=ram;
          ns:=n   ew:=0   c:=0;
          out:i   ram:=sm;

   }

}


move_n (in, out)
image in, out;

{  int i;

   for (i = 0;  i < size(in); i++)
   {
          in:i     ns:=ram;
          ns:=s   ew:=0   c:=0;
          out:i    ram:=sm;
   }

{

move_e (in, out)
image in, out;
{
   int i;
   for (i = 0; i < size(in); i++)
   {
          in:i     ew:=ram;
          ew:=w   ns:=0   c:=0
          out:i    ram:=sm;
   }

}

move_w (in, out)

image in, out;

{
    int i
    for (i = 0; i < size, i++)
```

```
        {
            ew:=in:i;
            ew:=e;   ns:=0;   c:=0;
            out:=sm;
        }
    }

    /*********************************************************
    **********************

    Procedure Name:  mux

    Purpose:  mux multiplexes one of two ram images A or B
    into a third image C depending on the value of the
    image sel.  If sel is a zero then A is moved into C
    otherwise B is moved into C.

    Image Variables:

        sel - the selection image
              for the multiplexer
        A   - the first selection
        B   - the second selection
        C   - the recipient image

    Integer Variables

        i   - used to count through all the bits of the
    selected image.

    Algorithm:  mux uses the following boolean formula for
    multiplexer:

              C = (A and (not(sel)) or (B and sel)

    Execution cycles:  7n - where n is the number of bits
    in the image.

    Author:  Robert E. Boring
    Revision:  1.0

    *********************************************************
    ************************/


mux (sel, A, B, C)

image sel, A, B, C;

{   int i;
    1* if sel = 0  then A-->C
       if sel = 1  then B-->C  *1
```

```
        1* form C = ((A) and (not(sel))) or ((B) and (sel))*1

           for (i = 0; i < size; i++)
           {
                1* form ((A) and (not(sel))) --> C register *1
                   A:i     ns:=sel   ew:=ram;
                           c:=bw;

                1* form ((B) and (sel)) --> C register *1
                   B:i     ew:=c     c:=ram;
                           c:=cy;

                1* form ((A) and (not(sel))) or ((B) and
        (sel)) --> bw
                    register *1
                    ns:=0;
                    c:=bw;
                    C:i     ram:=c;

            }

}
set_ram (A)

/*****************************************************************

Procedure Name:  set_ram

Purpose:  set_ram sets the LSB of a ram image to 1.  It clears all other
bits.

Image Variables:  A - input image to be set.

Integer Variables:  i - used to increment through all bits of
                    image A to perform the clearing operation.

Execution time: n + 3 - where n is the number of bits

Author:  Robert E. Boring

Revision:  1.0

*****************************************************************/


image A;

{
     int i;
     c:= 0

     for (i = 0;  i < size(A); i++)
         A:i     ram:=c;
```

```
      c:=1
      A:0      ram:=c;

}

/*****************************************************************

Procedure Name:  clear_ram

Purpose:  clear_ram sets all bits of an input image to zero

Image Variables:
     A - the image to be cleared

Integer Variables:
     i - increments through all bits of A

Execution Time:  n + 1 - where n is the number of bits in A.

Author:  Robert E. Boring

Revision: 1.0

*****************************************************************/


clear_ram (A)
image A;

{
     int:;
     c:=0;

     for  (i = 0; i < size(A); i++)
           A:i         ram:=c;

}

/*****************************************************************

Procedure Name:  move_ram

Purpose:  move_ram moves an input image A to an output image B at the
same processor cell.  If A is bigger than B then only the LSB's of A are
moved into B.  If A is smaller than B, then B is padded with zero's in
the MSB's.

Image Variables:

     A - input image
     B - output image

Integer Variables:
```

```
     i - size of A
     j - size of B
     count - increments through various bits of the images.

Execution Time:  if (A = > B) Zn where n is the number of bits in B
                 if (A < B) Zm + n + 1
                 where m size of A and n is size (B) - size (A)

Author:  Robert E. Boring
Revision: 1.0

*************************************************************************/

move_ram (A,B)
image A, B;

{
     int i = size (A);
     int j = size (B);
     int count = 0;

     for (count = 0; count < (i < = j?:i:j); count++)
     {

          A:count   c:=ram;
          B:count   ram:=c;
      }

      if (i < j)
      {
           c:=0
           for (count = i; count < j; count++)
                B:count    ram:=c;

       }
}
```

```
/*************************************************************************
*******

Procedure name:  chk_zero

Purpose:  chk_zero test the input image in to see if any bits in the
image
are set.  This equates to in being non-zero.  If in is non-zero, then
the
LSB of result is returned set to 1.  Otherwise, it is set to zero.

Image Variables:

   in - the input image to be tested
   result - stores the result of the test

Integer Variables:

   ; - counter for incrementing through the bits of the test image

Execution cycles:  2n + 2 - where n is the number of bits in the input
image

Author:  Robert E. Boring
Revision:  1.0

*************************************************************************
*****/

chK_zero (in, result)

image in, result;
{
      int:i;

      1*  if in is zero then a zero is placed in result otherwise a 1*/

      1*  test each bit of in by oring with result *1

      1*  use c to hold result *1

          c:= 0;  ns:=0;
          for (i = 0; i < size; i++)
          {

               in:i      ew:=ram;
                         c:=bw;
          }

      result:0    ram:=c;

}

/*************************************************************************
```

```
*******

Procedure Name:  conv

Purpose:  conv converts a signed magnitude number to two's complement or
vice-versa.  This operation is performed by complementing the LSB's of
the
input image if the MSB is a 1.  A one is then added if the MSB was 1.

Image Variables:

    x - input image to be converted

Integer Variables:

    n - the size of the input image
    i - a counter for bit sequencing of the input image

Execution Cycles:  4n-1 - where n is the bit length of the input image

Author:  Robert E. Boring
Revision:  1.0

*************************************************************************
******/

conv (X)
image X;

{
     int n = size (X);
     int i;

          X:n-1    c:=ram;

          1* if MSB (now in c) is a 1 the ew is inverted otherwise
             it is exclusive ored with ns which is zero *1

          for (i = 0; i < n - 1; i++)
          {

                  X:i      ew:=ram;
                  X:i      ram:=sm;
          }

          1* ADD 1 if MSB is 1*/

          for (i = 0; i < n; i++)
          {
                  X:i      ew:=ram;
                  x:i      ram:=sm       c:=cy;
          }

}
```

```
1* integer addition routine *1

/**************************************************************************
*******

Procedure Name:  add

Purpose:  add performs a signed magnitude addition operation on two
images.
The operation takes place by first converting the operands to two's
complement
form and then adding.  The operand lengths do not have to match.

Image Variables:

    A - operand one
    B - operand two
    R - result operand

Integer Variables:

    i - bit size of A
    j - bit size of B
    k - bit size of R
    counter - used for bit sequencing

Calls:  conv - converts signed magnitude image to two's complement image

Execution Cycles:  for equal sized operands 23n + 7

Author:  Robert E. Boring
Revision:  1.0

**************************************************************************
******/

add (A,B,R)

image A, B, R;

{

     int i = size (A);
     int j = size (B);
     int k = size (result);
     int counter = 0;

     conv(A);     *1 form 2's comp from signed image *1
     conv(B);

     A:0    ew:=ram;
     B:0    ns:=ram;
     R:0    ram:=sm      c:=cy;
```

```
    counter = 0;

    1* ADD *1

    while ( counter < (i < j?:j:i))
    {

        A:counter       ew:=ram;
        B:counter       ns:=ram;
        R:counter       ram:=sm       c:=cy;

        counter += 1;
    }

    if  (i <= j)  A:i - 1     ns:=ram;  1* Sign extend A *1
    if  (j <= i)  B:j - 1     ew:=ram;  1* Sign extend B *1

    counter = (i < j?:i:j)
    while (counter < ( i > j?:i:j))
    {
        if ( i > j )      A:counter     ns:=ram;
        if ( j > i )      B:counter     ew:=ram;

        R:counter     ram:=sm    c:=cy;

        counter += 1;

     }

     conv (A);     1* convert 2's comp to signed magnitude *1
     conv (B);
     conv (R);

}

    1* fractional signed magnitude multiply *1

/*****************************************************************************
*******

Procedure Name:  mult

Purpose:  mult performs a signed magnitude multiply on two images.  The
operand
lengths do not have to match.

Image Variables:

    temp - used to hold temporary partial products
    A - multiplicand input
    B - multiplier input
    R - product output

Integer Variables:
```

```
    m - bit length of A
    n - bit length of B
    o - bit length of R
    i,j - bit sequencer counters

Calls:  clear_ram - clears some of the image variables before multiply
proceeds

Execution Cycles:  If A and B operands are equal size:
                   6n^2 + n + 10 - where n is the bit length of the
                   input operands

Author:  Robert E. Boring
Revision:  1.0

**************************************************************************
******/

mult (A, B, R)
image A, B, R;

{
     image temp1 : size (A) + size (B) - 2
     int  m = size (A);     int g = size (temp)
     int  n = size (B);
     int  o = size (R);
     image scratch : 1;
     int i,j;

     clear_ram (R);
     clear_ram (scratch);
     clear_ram (temp1);

     for ( i = 0; i < n-2; i ++)   1* for each bit of B *1
     {
          for (j = 0; j < m-2; i ++)  1* for each bit of A *1
          {
               B : i  ew := ram;
               A : j  ns := ram  c := 0;  1* AND the bits *1

               temp : (j + i)  ns: = ram  c := cy;  1* add current R
                                                    to AND products *1
               scratch : o     ew:= ram      1* recall last carry *1

               temp (j + i)   ram:= sm   c:= cy;
               scratch : o    ram: = c;
          }
          clear_ram (scratch);
          temp ( j + i + 1)    ram:= c
     }

     1* AND the sign bits and place as sign bit of temp *1

     A : (m - 1)     ns := ram;
```

```
     B : (n - 1)      sw := ram;      c := o;
     temp : (g - 1)   ram := sm;

1* copy MSB's of temp to R *1

     for (i = 0 ; i < (g < o ? g : o); i ++)
     {

          temp : (g-i)  c := ram;
          R    : (o-1)  ram := c;

     }

1* if temp is smaller than R then pad MSB's of R *1

   if (g < o)
   {
       c := o;
       for ( i = o; i < (o-g) ; i++)
            R : i     ram := c;
    }

}     1* END *1
```

```
/*************************************************************************
*******

Procedure Name:  calc_boundary

Purpose:  calc_boundary performs the boundary cell emulation from
Whirter's
least squares minimization algorithm.  The equations solved by this cell
are
as follows:

    If (XI = 0 or DI = 0) then
    {
        CO=1;
        SI=0;
        Dout=Din;
        Z=XI;
    }
    else
    {
        DP = B2d + DIXI2;
        CO = B2d/dP;
        SI = DIXI/dP;
        Z  = XI;
        d  = dP
        DO = COSI;
    }

Image Variables:
```

```
    XI,DI,CO,SI,Z,DO,dP,B,d,temp1,temp2,Flag1,Flag2 - see declarations
    in main program

Calls:  chk_zero
        set
        mux
        clear_ram
        mult
        add
        div
        sub

Execution Cycles:  60n^2 + 135n + 119 - where n is the bit lengths of the
                   major operands XI,DI,DO,CO,SI,Z,dP,B,d

Author:  Robert E. Boring
Revision:  1.0

*************************************************************************
******/

calc _ boundary ( XI, DI, CO, SI, Z, DO, dP, d, B)

image  XI, DI, CO, SI, Z, DO, dP, d, B ;

{

     chk _ zero ( XI, Flag1);
     chk _ zero ( DI, Flag2);

     1* check Flag 1 and Flag 2, if either is zero then a zero is placed
in Flag 1 *1

    c := 0;
    ns := Flag 1 ;
    ew := Flag 2 ;

     c := cy ;

    Flag1 := c ; 1* put result in Flag1*/
    /* generate a 1 in temp1*/
          set (temp1)
          mux (Flag1, temp1, C, temp2)
          mux (boundary_mask, temp2, CO, CO)
          clear_ram (temp1)
          mux ( Flag 1, temp1, SI, temp2)
          mux (boundary_mask, temp2, SI, SI)
          mux (boundary_mask, XI, Z, Z)
          mux (boundary_mask, DI, DO, DO)

     1* calculate  dP *1
          mult (XI, XI, temp1)
          mult (B, B, temp2)
          mult (temp2, d, temp2)
```

```
            mult (DI, temp1, temp1)
            add  (temp1, temp2, temp1)
            mux  (Flag1, dP, temp1, temp1)
            mux  (boundary_mask, temp1, dP, dP)

        1* calculate  CO *1
            mult (B, B, temp1)
            mult (temp1, d, temp1)
            div  (d, dP, temp1)
            mux  (Flag1, CO, temp1, temp1)
            mux  (boundary_mask, temp1, CO, CO)

        1* calculate  SI *1
            mult (DI, XI, temp1)
            div  (temp1, dP, temp1)
            mux  (Flag1, SI, temp1, temp1)
            mux  (boundary_mask, temp1, SI, SI)

        1* calculate  d *1
            mux  (Flag1, d, dP, temp1)
            mux  (boundary_mask, temp1, d, d)

        1* calculate  DO *1
            mult (C, DI, temp1)
            mux  (Flag1, DO, temp1, temp1)
            mux  (boundary_mask, temp1, DO, DO);

        1* move DO one south *1
           move_s (DO, DO)

}
/* End of Calc_boundary */

/*****************************************************************************
*******

Procedure Name:  calc_interval

Purpose:  calc_interval performs the internal cell emulation of the
Whirter's least squares minimization algorithm.  The equations solved by
the internal cell are:

    XO = XI - ZR;
    R  = COR + SIXI;

Image Variables:

    XI,XO,DI,DO,R,LO,SI,Z,d,dP,B,temp1,temp2 - see main program
documentation

Calls:  mult
        sub
        mux
        add
```

```
Execution Cycles:  18n^2 + 63n + 7 - where n is the bit length of the
major
                   operands XI,XO,DI,DO,SI,CO,Z,R,d,dP,R.

Author:  Robert E. Boring
Revision:  1.0

**************************************************************************
******/

calc_internal ( XI, XO, DI, DO, R, CO, SI, Z, d, dP, B);
image  XI, XO, DI, DO, R, C, S, Z, d, dP, B;

{

          mult (Z, R, temp1)
          sub (XI, temp1, temp1)
          mux  (internal_mask, temp1, XO, XO)
          mult (C, R, temp1);
          mult (S, XI, temp2);
          add  (temp1, temp2, temp1);
          mux  (internal_mask, temp1, R, R)

}

/**************************************************************************
********
Procedure Name:  calc_final

Purpose:  calc_final performs the final-cell emulation of Whirter's
least
squares minimization solution.  The equations solved by the final cell
are:

     XO = DI XI;

Image Variables:

    XI,DI,temp1,final_mask - see main program for documentation

Calls:  mult
        mux

Execution Cycles:  6n^2 + 8n + 10 - where n is the bit length of the
major
                   operand XI,DI,XO,DO.

Author:  Robert E. Boring
Revision:  1.0

**************************************************************************
****/

calc_final (DI, XI)
```

```
{

        mult (DI, XI, temp1)
        mux  (final_mask, temp1, XO, XO)

}
```

APPENDIX C

ARRAY PROCESSING ARCHITECTURE SIMULATOR GENERATOR

In the course of this SBIR research, a zero-order simulator generation system for systolic array and hypercube simulation was produced. The generator allows specification of an n-dimensional array of processors with user defined attributes. The generator then generates a simulator in the C programming language to simulate the specified computational network. Instructions for the writing of a simulator specification are given in the documentation header of the generator code file. Appendix D gives an example simulator specification for performing a Givens rotation. Appendix E shows the execution results on sample data.

The simulator is simplistic. The burden for providing input/output functions as well as functions for the internal operation of the PE's is placed on the user. With more development, these tasks could be shifted to the simulator generator. A more sophisticated specification grammar and translator would make this generator a truly useful tool for testing highly parallel algorithms such as those found in adaptive beamforming. The major goal in the creation of this generator was to demonstrate the feasibility of design and simulation tools for array processing in general and, more importantly, to provide a testbed for the adaptive tracking and beamforming concepts studied.

```
/**********************************************************************
ROUTINE OR FILE NAME: gensyssim.c

PURPOSE: gensyssim.c generates an array processor simulation program
from user specifications of the array processor.  The current system
was generated as part of a DOD study on adaptive beamform solutions.
In this version input of user specifications must obey very strict
lexical and syntactic conventions.  These restrictions are due to the
lack of lexical and parser generator tools which could be used to
improve the user friendliness and the power of this program.

        The program operates by executing the following command:

                gensyssim inputfile outputfile

        The program reads the simulator specification from the inputfile
and produces a C program in outputfile.  The C program is then compiled
and linked to produce an executable simulator.

        The inputfile must have the following format:

        dimnumber           - the number of processor dimensions declared
        dimension list      - this is a list of numbers(one per line)
                              which defines the size of each dimension
                              There must be as many numbers in this list
                              as there are dimensions declared in the
                              first number of this file
        display time step-  - this number defines the number of execution
                              time steps which occur between displays
        pe number           - this number tells the generator how many different
                              pe types have been declared
        pefilename          - this is the name of the file that contains the
                              C routines which define each of the pe's
        variable list       - this is a list of variables which the user
                              needs to be visible at each array element.
                              These declarations must conform to C declartions
                              that is they must have the form:
                                   type  variable
                              where type is a C type and variable is the variable
                              name.


        The simulator adopts some conventions which must be discussed.
First the user must write C subroutines which define the operation of various
functions of the simulator such as input/output, ect. The following functions
must be defined by the user at this time.  Automatic generation of some or all
of these functions could be handled given a sufficiently powerful parser.  Auto-
matic generation of all routines including PE definition is suggested for a
full blown simulator.

        USER FUNCTIONS:
```

PE definition functions	- declarations of the form given below which define the operation of each PE
initialization function	- a routine which sets up initial array contents, opens user input and output files, and any other assorted initialization tasks.
input function	- controls input from the user input file to the array.
output function	- controls user output from the array.
exchange function	- causes the contents of the "IN" array to exchange with the "OUT" array.
display function	- a function to display any array contents desired at the end of the number of "execution cycles" given by the display time step set in the simulator specification file.

A more detailed discussion of each of the required routines follows:

PE declarations:

Each PE is a C subroutine with the structure:

```
pe#()
{
        user declarations and statements
}
```

where "#" is the pe type number(numbering must begin with 0).

The subroutines have access to global arrays where information about the input output state of the array is maintained. These arrays are "IN" and "OUT" and are dimensioned according to the user's dimensions in the inputfile.

"IN" and "OUT" are arrays of structures containing the user's variables declared in the inputfile to the generator. It is assumed that the array "IN" contains the results of the last iteration of all pe's. "OUT" is used to hold output results from the operation of the pe on this iteration. This convention must be held if simultaneous operation of all pes is to be maintained. This is a sequential program simulating concurrent operation of pes, therefore the actual execution order of the pes is not guaranteed. The user should then assume that the only results available to him from other pes comes from the "IN" array. Additionally the user should not modify his own or another pe's "IN" array, results should be written to the out array. The user may declare as many temporary variables within the pe subroutines as desired. When all pes have been run the main driver assigns the results of the "OUT" array to the "IN" array. Any current "IN" array information required during the next iteration must be copied to the "OUT" array.

A series of global variables of the form "COUNT#" where # is a dimension number are also available. During the execution of the simulator these variables define the currently executing pe. The pe subroutine should use these variables to access the "IN" and "OUT" array for the pe. When accessing

information from other pes a local communication limit of plus or minus one pe in any dimension should be adhered to. If this is not adhered to a run time error will result.

The user variables are accessed in the following form:

```
IN[COUNT1][COUNT2][COUNT3][...].user
        or
OUT[COUNT1[COUNT2][COUNT3][...].user
```

where "user" is a user variable name declared in the specification inputfile.

initialization function:

The initilization function is a subroutine of the form:

```
INIT()
{
        user declarations and statements.
}
```

This function is called by the simulator as the first action it performs. This function is responsible for initializing the IN and OUT arrays as well as the PETYPE array which defines where different PE types occur in the processor array. INIT() should also be responsible for opening the user input and user output files. Two file pointers are available for this, USERINPUT and USEROUTPUT. The INIT() function has access to the variables of the form COUNT# and the constants of the form DIM# (These are discussed above under PE declarations).

input function:

The input function must have the form:

```
INPUT()
{
        user declarations and statements
}
```

The input function is called before every execution of the processors. The function is provided so that the user may read input from some source and load it into the appropriate locations of the IN array. A typical example of this function would be one that reads a line of numbers from a disk file and places it on some edge of the IN array. This type of input occurs in matrix multiplication and other algorithms.

output function:

An output function of the form:

```
OUTPUT()
{
        user declarations and statements
}
```

must be supplied. This function is called after every execution of the processor array and before the exchange function is called. This function is used to perform any desired output from the array results to a device such as the disk.

exchange function:

The exchange function must have the form:

```
EXCHANGE()
{
        user declarations and statements
}
```

The exchange function is called following the execution of the processor array. This function must move the contents of the OUT array to the IN array for the next cycle of execution. Thus the statements will typically be of the form:

```
IN[COUNTO][...].user_var = OUT[COUNTO][...].user_var;
```

All output variables that are required as input for any PE type must be copied in this manner.

display function:

The display function must have the form:

```
DISPLAY()
{
        user declarations and statements
}
```

The display function is called after a number of processor cycles equal to the number given by the display time step in the simulator specification file. The purpose of this function is to provide the user with some display of the array state after a set of equally separated processor execution cycles. The function may perform screen or file output.

All of these functions must be declared, however, any or all of them may contain no functional code as the simulator user may not require some of the functions.

Time has not permitted as much automation as can be achieved in for this simulator. With sufficient power in the lexical analyser/parser for the simulation specification file, variable names could be retained for use in the automatic generation of default functions to perform the exchange operation and the display operation. With an expansion of the specification syntax automatic generation of the PE's, input/output, and initilization could be established. These revesions are recommended as part of the developement of a more powerful simulation tool for hypercube algorithms.

```
**********VARIABLE DICTIONARY**********

GLOBALS:

LOCALS:

INPUT:

OUTPUT:

CALLED BY:

CALLS:

AUTHOR: Robert E. Boring

REVISION: 1.0 January 16,1986

**********/
#include <stdio.h>
#define NULL 0
#define EOF -1

main(argc,argv)
int argc;
char **argv;
{
FILE *INPUTFILE,*OUTPUTFILE,*DISPFILE,*fopen();
int *pnumdim,*pcurdimnum,numdim,curdimnum,i;
int *pdispnum,dispnum;
int *ppenum,penum;
char pefilename[32],*pvardeclname,vardeclname;
pvardeclname = &vardeclname;
pnumdim = &numdim;
pcurdimnum = &curdimnum;
pdispnum =&dispnum;
ppenum = &penum;

/*OPEN ALL FILES KNOWN AT THIS TIME*/
        if(argc!=3)
        {
                printf("Usage: gensyssim inputfile outputfile\n");
                exit(1);
        }


        if((INPUTFILE = fopen(argv[1],"r")) == NULL)
        {
                printf("can't open %s\n",argv[1]);
                exit(1);
        }

        if((OUTPUTFILE = fopen(argv[2],"w")) == NULL)
```

```
        {
                printf("file creation error: %s\n",argv[2]);
                exit(1);
        }


/*BEGIN GENERATION OF C CODE*/
        fprintf(OUTPUTFILE,"#include <stdio.h>\n");

/*READ THE NUMBER OF DIMENSIONS*/
        if(fscanf(INPUTFILE,"%d",pnumdim) == EOF)
        {
                printf("premature end of file, missing number of dimensions\n");
                exit(1);
        }

/*GENERATE #DEFINE NUL AND #DEFINE EOF*/
        fprintf("#define NULL 0\n#define EOF -1\n\n");

/*GENERATE DIMENSION CONSTANTS CODE*/
        for(i=0;i<numdim;i++)
        {
                if(fscanf(INPUTFILE,"%d",pcurdimnum) == EOF)
                {
                        printf("premature end of file, not enough
dimension arguments\n");
                        exit(1);
                }
                fprintf(OUTPUTFILE,"#define DIM%d %d\n",i,curdimnum+2);
        }

/*READ THE DISPLAY TIME STEP*/
                if(fscanf(INPUTFILE,"%d",pdispnum) == EOF)
                {
                        printf("premature end of file, display time missing\n);
                        exit(1);
                }
                fprintf(OUTPUTFILE,"#define D_TIME_STEP %d\n\n",dispnum);

/*READ NUMBER OF DIFFERENT PE TYPES*/
                if(fscanf(INPUTFILE,"%d",ppenum) == EOF)
                {
                        printf("premature end of file,  number of
pe types missing\n");
                        exit(1);
                }

/*READ PE FILE NAME*/

                if(fscanf(INPUTFILE,"%s",pefilename) == EOF)
                {
                        printf("premature end of file, pe file name missing\n");
                        exit(1);
```

```
                }

/*READ VARIABLE DECLARATIONS AND CREATE INPUT/OUTPUT STRUCTURE DECLARATION
CODE*/
                fprintf(OUTPUTFILE,"struct INOUTSTRUCT\n");
                fprintf(OUTPUTFILE,"{\n");

                while(fscanf(INPUTFILE,"%c",pvardeclname) != EOF)
                {
                        if(vardeclname == '\n')
                                fprintf(OUTPUTFILE,"\n\t");
                        else
                                fprintf(OUTPUTFILE,"%c",vardeclname);
                }

                fprintf(OUTPUTFILE,"\n}\n");


/*GENERATE "IN" AND "OUT" ARRAY DECLARATIONS*/
                fprintf(OUTPUTFILE,"IN");
                for(i=0;i<numdim;i++)
                        fprintf(OUTPUTFILE,"[DIM%d]",i);

                fprintf(OUTPUTFILE,",\nOUT");
                for(i=0;i<numdim;i++)
                        fprintf(OUTPUTFILE,"[DIM%d]",i);
                fprintf(OUTPUTFILE,";\n\n");

/*GENERATE PE TYPE ARRAY DECLARATION*/
                fprintf(OUTPUTFILE,"int PETYPE");
                for(i=0;i<numdim;i++)
                        fprintf(OUTPUTFILE,"[DIM%d]",i);
                fprintf(OUTPUTFILE,";\n\n");

/*GENERATE ARRAY ADDRESSING COUNTERS DECLARATION*/
                for(i=0;i<numdim;i++)
                        fprintf(OUTPUTFILE,"int COUNT%d;\n",i);

/*GENERATE DCOUNT DECLARATION*/
                fprintf(OUTPUTFILE,"int DCOUNT;\n\n");

/*GENERATE USER INPUT FILE DECLARATIONS IN CASE THE USER WANTS TO
RUN SOME INPUT FUNCTION*/

                fprintf(OUTPUTFILE,"FILE *USERINPUT,*USEROUTPUT, *fopen();\n

/*GENERATE INCLUDE STATEMENT FOR THE USER PE FILE*/
                fprintf(OUTPUTFILE,"#include <%s>\n\n",pefilename);

/*GENERATE PAUSE FUNCTION*/
                fprintf(OUTPUTFILE,"PAUSE()\n");
                fprintf(OUTPUTFILE,"{\n\tchar c;\n");
                fprintf(OUTPUTFILE,"\tprintf(\"\\nStrike   N   to
```

```
Exit Any Other Key to Continue\\n\");\n");
                fprintf(OUTPUTFILE,"\twhile(!kbhit());\n");
                fprintf(OUTPUTFILE,"\tc = getch();\n");
                fprintf(OUTPUTFILE,"\tif((c == 'n')||(c == 'N'))\n");
                fprintf(OUTPUTFILE,"\t\texit(1);\n");
                fprintf(OUTPUTFILE,"}\n\n");


/*GENERATE MAIN PROGRAM*/

        fprintf(OUTPUTFILE,"main()\n{\n");
        fprintf(OUTPUTFILE,"\tINIT();\n");
        fprintf(OUTPUTFILE,"\twhile(1)\n\t{\n");
        fprintf(OUTPUTFILE,"\tfor(DCOUNT=0;DCOUNT<D_TIME_STEP;DCOUNT++)\n\t{\n"
        fprintf(OUTPUTFILE,"\t\tINPUT();\n");

/*GENERATE AS MANY FOR LOOPS AS THERE ARE DIMENSIONS*/
        for(i=0;i<numdim;i++)
                fprintf(OUTPUTFILE,"\t\tfor(COUNT%d=1;COUNT%d<DIM%d-1;COUNT%d++)

/*GENERATE SWITCH STATEMENT FOR SELECTION OF PE TYPE*/
        fprintf(OUTPUTFILE,"\t\t\tswitch(PETYPE");
        for(i=0;i<numdim;i++)
                fprintf(OUTPUTFILE,"[COUNT%d]",i);
        fprintf(OUTPUTFILE,")\n\t\t\t{\n");

/*GENERATE CASE STATEMENTS AND PE CALLS*/
        for(i=0;i<penum;i++)
                fprintf(OUTPUTFILE,"\t\t\t\tcase %d:{pe%d();break;}\n",i,i);
        fprintf(OUTPUTFILE,"\t\t\t\tdefault:{printf(\"pe undefined at PETYPE\");
        fprintf(OUTPUTFILE,"printf(\"");
        for(i=0;i<numdim;i++)
        {
                fprintf(OUTPUTFILE,"[%c",'%');
                fprintf(OUTPUTFILE,"d]");
        }

        fprintf(OUTPUTFILE,"\"");
        for(i=0;i<numdim;i++)
                fprintf(OUTPUTFILE,",COUNT%d",i);
        fprintf(OUTPUTFILE,");}\n");

        fprintf(OUTPUTFILE,"\t\t\t}\n\n");

/*GENERATE CODE TO COPY OUTPUT VARIABLES TO THE INPUT ARRAY*/
        fprintf(OUTPUTFILE,"\t\tOUTPUT();\n");
        fprintf(OUTPUTFILE,"\t\tEXCHANGE();\n");

        fprintf(OUTPUTFILE,"\t}\n\tDISPLAY();\n");
        fprintf(OUTPUTFILE,"\tPAUSE();\n\t}\n}\n");


fclose(INPUTFILE);
```

```
fclose(OUTPUTFILE);
}
```

Appendix D
Square-root Free Givens Rotation Simulator

```
/*This file contains the Givens rotation simulator specification parameters.
See documentation header of GENSYSIM for an explanation of each of these
parameters*/

2
3
3
1
4
pefile.h
float x;
float c;
float s;
float d;
float delta;
float z;
float beta;
float r;
```

```
/*************************************************************************
ROUTINE OR FILE NAME: pefile.h

PURPOSE:        This file declares the subroutines necessary to simulate
the squareroot free Givens rotation algorithm developed by McWhirter. The
processor elements and associated simulator support subroutines are defined in
this file.

************************VARIABLE DICTIONARY*****************************

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24,1986

Copyright 1986 by Space Tech Corporation.  Fort Collins CO, USA.
All rights reserved.

*************************************************************************/
```

```
/*************************************************************************
ROUTINE OR FILE NAME: pe0

PURPOSE:  pe0 simulates the inactive cells in a rectilinear array which is
simulating McWhirter's triangular Givens rotation array.  The computationally
inactive processors simply move the x variable of the processor immediately
north of the current processor to current processor position.
************************VARIABLE DICTIONARY*****************************

GLOBALS: int COUNT0 - the current x location of the processor
         int COUNT1 - the current y location of the processor
         INOUTSTRUCT IN - the input array of data which defines the current
                          processing array state
         INOUTSTRUCT OUT - the output array of data which defines the modified
                           next state of the array after the current processor
                           has executed.


CALLED BY: The automatically generated simulator created by program gensyssim

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24, 1986

*************************************************************************/

pe0()
{
        OUT[COUNT0][COUNT1].x = IN[COUNT0][COUNT1-1].x;
}
```

```
/*****************************************************************
ROUTINE OR FILE NAME: pe1

PURPOSE:  This subroutine simulates the operation of the boundary cell
of McWhirter's Given's rotation algorithm.  This cell generates the
sine and cosine terms which are broadcast to the rest of the array.

              Algorithm:

                     if x or delta = 0 then
                             cosine = 1;
                             sine   = 0;
                             delta (out) = delta (in)
                             z = x
                     else
                             dp = beta * d  +  delta (in) * x (in) * x (in)
                             cosine = beta * d / dp
                             sine = delta (in) * x (in) /dp
                             z = x (in)
                             d = dp
                             delta (out) = cosine * sine


**********************VARIABLE DICTIONARY*****************************

GLOBALS: int COUNTO - location of the current PE x dimension
         int COUNT1 - location of the current PE y dimension
         INOUTSTRUCT IN - the input array, defines the current processor array
                          state on this execution cycle
         INOUTSTRUCT OUT - the output array, defines the results of this
                           execution of the processor array


LOCALS: float x - holds the value of the x input which is obtained from the
                  processor immediately to the north of the current processor
        float delta - used to hold the delta variable which is obtained from
                      the processor diagonal (up and left) of the current
                      processor
        float d     - temporary used to hold the d variable which is obtained
                      from the current processor
        float beta  - hoold the beta variable obtained from the currrent
                      processor


CALLED BY: the simulator generated by gensyssim

CALLS: none

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24,1986

*****************************************************************/

pe1()
```

```
/*************************************************************************
ROUTINE OR FILE NAME: pe2

PURPOSE: pe2 simulates the operation of McWhirter's givens rotation
internal cell.  The equations processed are:

                x (out) = x (in) - z (in) * r
                r (out) = c (in) * r (in) + s (in) * x (in)


************************VARIABLE DICTIONARY*****************************

GLOBALS: int COUNTO - indicates the current processor location x dimension
         int COUNT1 - indicates the current processor location y dimension
         INOUTSTRUCT IN - the input array defining the current array input
                          state
         INOUTSTRUCT OUT - the output array defining the array output state
                           after the PE has executed

LOCALS: float x - temporaries used to hold array variables during intermediate
        float c   computation
        float s
        float z
        float r


CALLED BY: simulator routine generated by gensyssim

AUTHOR:  Robert E. Boring

REVISION: 1.0 January 24,1986

*************************************************************************/

pe2()
{
float r,x,c,s,z;

x = IN[COUNTO][COUNT1-1].x;
c = IN[COUNTO-1][COUNT1].c;
s = IN[COUNTO-1][COUNT1].s;
z = IN[COUNTO-1][COUNT1].z;
r = IN[COUNTO][COUNT1].r;
        OUT[COUNTO][COUNT1].x = x-z*r;
        OUT[COUNTO][COUNT1].r = c*r+s*x;
        OUT[COUNTO][COUNT1].s = s;
        OUT[COUNTO][COUNT1].c = c;
        OUT[COUNTO][COUNT1].z = z;
}
```

```
/*************************************************************************
ROUTINE OR FILE NAME: pe3

PURPOSE: pe3 simulates the final cell in McWhirter's algorithm.

                The final cell realizes just one equation:

                        x (out) = x (in) * delta

************************VARIABLE DICTIONARY*****************************

GLOBALS: int COUNTO - indicates the current processor location x dimension
         int COUNT1 - indicates the current processor location y dimension
         INOUTSTRUCT IN - the input array defining the current array input
                          state
         INOUTSTRUCT OUT - the output array defining the array output state
                           after the PE has executed

LOCALS: float x - temporaries used to compute the pe equations
        float delata

CALLED BY: simulator routine generated by gensyssim

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24, 1986

*************************************************************************/

pe3()
{
float x,delta;

delta = IN[COUNTO-1][COUNT1-1].delta;
x = IN[COUNTO][COUNT1-1].x;

        OUT[COUNTO][COUNT1].x = delta*x;
}
```

```
/*****************************************************************
ROUTINE OR FILE NAME: Init

PURPOSE:  init initializes all of the global variables for the McWhirter
Givens rotation.  The variables of the arrays IN and OUT are initialized
to 0 except the beta array which is initialized to 1 since it is a
weighting matrix.  The PETYPE array is initialized to represent the PE
arrangement of McWhirter's triangular array when implemented on a rectilinear
array.  init also opens the user input and output data files which are
named input.dat and output.dat on the default disk.

**********************VARIABLE DICTIONARY*****************************

GLOBALS: INOUTSTRUCT IN - the input array defining the current array input
                          state
         INOUTSTRUCT OUT - the output array defining the array output state
                           after the PE has executed
         int PETYPE - the array defining the PE positions in the rectilinear
                      array
         FILE *USERINPUT - the file pointer to the user input file
         FILE *USEROUTPUT -  the file pointer to the user output file

LOCALS: XCOUNT , YCOUNT - counters to address the 2 - dimensional arrays

CALLED BY: simulator generated by gensyssim


AUTHOR: Robert E. Boring

REVISION: 1.0 January 24, 1986

*****************************************************************/


INIT()
{
int XCOUNT,YCOUNT;
        for(XCOUNT=0; XCOUNT<DIM0; XCOUNT++)
                for(YCOUNT=0; YCOUNT<DIM1; YCOUNT++)
                {
                        IN[XCOUNT][YCOUNT].beta = 1;
                        IN[XCOUNT][YCOUNT].x = 0;
                        OUT[XCOUNT][YCOUNT].x = 0;
                        IN[XCOUNT][YCOUNT].r = 0;
                        IN[XCOUNT][YCOUNT].s = 0;
                        IN[XCOUNT][YCOUNT].c = 0;
                        IN[XCOUNT][YCOUNT].d = 0;
                        IN[XCOUNT][YCOUNT].delta = 0;
                        IN[XCOUNT][YCOUNT].z = 0;
                        if((XCOUNT == 0)||(YCOUNT == 0)||(XCOUNT == (DIM0-1))||
                                        (YCOUNT == (DIM1-1)))
                                PETYPE[XCOUNT][YCOUNT] =0;
                        else if((XCOUNT==(DIM0-2))&&(YCOUNT==(DIM1-2)))
                                PETYPE[XCOUNT][YCOUNT] = 3;
                        else if(XCOUNT == YCOUNT)
```

```
                        PETYPE[XCOUNT][YCOUNT] = 1;
                    else if(XCOUNT<YCOUNT)
                        PETYPE[XCOUNT][YCOUNT] = 0;
                    else
                        PETYPE[XCOUNT][YCOUNT] = 2;
                }
        if((USERINPUT = fopen("input.dat","r")) == NULL)
        {
                printf("can't open input.dat\n");
                exit(1);
        }
        if((USEROUTPUT = fopen("output.dat","w")) == NULL)
        {
                printf("can't open output.dat\n");
                exit(1);
        }
}
```

```
/*************************************************************************
ROUTINE OR FILE NAME: display

PURPOSE:        display merely prints the contents of the IN array x
variable.

************************VARIABLE DICTIONARY*****************************

GLOBALS: INOUTSTRUCT IN - the input array defining the current array input
                          state

OUTPUT: printed contents of the IN array

CALLED BY: simulator generated by gensyssim

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24, 1986

*************************************************************************/

DISPLAY()
{
int XCOUNT,YCOUNT;
        printf("\n");
        for(YCOUNT=0; YCOUNT<DIM1; YCOUNT++)
        {
                for(XCOUNT=0; XCOUNT<DIM0; XCOUNT++)
                        printf("%f  ",IN[XCOUNT][YCOUNT].x);
                printf("\n");
        }
        printf("\n");
}
```

```
/*************************************************************************
ROUTINE OR FILE NAME: input

PURPOSE: input reads input from the file pointed to by USERINPUT into
the top edge of the IN array.  x variable values are read from the file.
Input also continually inputs a 1 in the delta variable of
the IN array at position 0,0; this is consistant with McWhirter's algorithm.

************************VARIABLE DICTIONARY*****************************

GLOBALS: INOUTSTRUCT IN - the input array defining the current array input
                          state

LOCALS: float *pnum - pointer to the floating point number for input
        float num - the floating point input number
        int i - used for sequencing the IN array

CALLED BY: simulator routine generated by gensyssim

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24, 1986

*************************************************************************/

INPUT()
{
float *pnum,num;
int i;
pnum = &num;

        fprintf(USEROUTPUT,"USER INPUT\n");
        for(i=1;i<DIMO-1;i++)
        {
                fscanf(USERINPUT,"%f",pnum);
                IN[i][0].x = num;
                fprintf(USEROUTPUT,"%f ",num);
        }
        fprintf(USEROUTPUT,"\n\n");
        IN[0][0].delta = 1;
}
```

```
/*************************************************************************
ROUTINE OR FILE NAME: OUTPUT

PURPOSE: output is responsible for outputing the processor contents
on every cycle.  This routine outputs the x variable, the sine and
cosine coefficients, the r variable, and the z coefficient.

************************VARIABLE DICTIONARY*****************************

        INOUTSTRUCT OUT - the output array defining the array output state
                          after the PE has executed
        FILE *USEROUTPUT -  the file pointer to the user output file

LOCALS: XCOUNT , YCOUNT - counters to address the 2 - dimensional arrays

OUTPUT: output to file of the output state variables detailed above.

CALLED BY: simulator generated by the routine gensyssim

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24, 1986

*************************************************************************/

OUTPUT()
{
int XCOUNT,YCOUNT;
        fprintf(USEROUTPUT,"\nSIN contents\n");
        for(YCOUNT=1; YCOUNT<DIM1-1; YCOUNT++)
        {
                for(XCOUNT=1; XCOUNT<DIMO-1; XCOUNT++)
                        fprintf(USEROUTPUT,"%f  ",OUT[XCOUNT][YCOUNT].s);
                fprintf(USEROUTPUT,"\n");
        }
        fprintf(USEROUTPUT,"\n");

        fprintf(USEROUTPUT,"\nCOSINE contents\n");
        for(YCOUNT=1; YCOUNT<DIM1-1; YCOUNT++)
        {
                for(XCOUNT=1; XCOUNT<DIMO-1; XCOUNT++)
                        fprintf(USEROUTPUT,"%f  ",OUT[XCOUNT][YCOUNT].c);
                fprintf(USEROUTPUT,"\n");
        }
        fprintf(USEROUTPUT,"\n");

        fprintf(USEROUTPUT,"\nZ contents\n");
        for(YCOUNT=1; YCOUNT<DIM1-1; YCOUNT++)
        {
                for(XCOUNT=1; XCOUNT<DIMO-1; XCOUNT++)
                        fprintf(USEROUTPUT,"%f  ",OUT[XCOUNT][YCOUNT].z);
                fprintf(USEROUTPUT,"\n");
        }
        fprintf(USEROUTPUT,"\n");
```

```
        fprintf(USEROUTPUT,"\nd contents\n");
        for(YCOUNT =1; YCOUNT<DIM1-1;YCOUNT++)
        {
                for(XCOUNT=1; XCOUNT<DIMO-1;XCOUNT++)
                        fprintf(USEROUTPUT,"%f ",OUT[XCOUNT][YCOUNT].d);
                fprintf(USEROUTPUT,"\n");
        }
        fprintf(USEROUTPUT,"\n");

        fprintf(USEROUTPUT,"\nX contents\n");
        for(YCOUNT=1; YCOUNT<DIM1-1; YCOUNT++)
        {
                for(XCOUNT=1; XCOUNT<DIMO-1; XCOUNT++)
                        fprintf(USEROUTPUT,"%f  ",OUT[XCOUNT][YCOUNT].x);
                fprintf(USEROUTPUT,"\n");
        }
        fprintf(USEROUTPUT,"\n");

        fprintf(USEROUTPUT,"\nR contents\n");
        for(YCOUNT=1; YCOUNT<DIM1-1; YCOUNT++)
        {
                for(XCOUNT=1; XCOUNT<DIMO-1; XCOUNT++)
                        fprintf(USEROUTPUT,"%f  ",OUT[XCOUNT][YCOUNT].r);
                fprintf(USEROUTPUT,"\n");
        }
        fprintf(USEROUTPUT,"\n");

        fprintf(USEROUTPUT,"RESIDUAL: %f",OUT[DIMO-2][DIM1-2].x);
        fprintf(USEROUTPUT,"");
}
```

```
/**************************************************************************
ROUTINE OR FILE NAME: Exchange

PURPOSE: Exchange copies the output state in array OUT to the input
state array IN.  It copies all variables.

************************VARIABLE DICTIONARY*****************************

GLOBALS: INOUTSTRUCT IN - the input array defining the current array input
                          state
         INOUTSTRUCT OUT - the output array defining the array output state
                           after the PE has executed

LOCALS: COUNTO , COUNT1 - counters to address the 2 - dimensional arrays

CALLED BY: simulator generated by gensyssim

AUTHOR: Robert E. Boring

REVISION: 1.0 January 24, 1986

**************************************************************************/

EXCHANGE()
{
for(COUNTO=0;COUNTO<DIM0;COUNTO++)
for(COUNT1=0;COUNT1<DIM1;COUNT1++)
{
     IN[COUNTO][COUNT1].beta=OUT[COUNTO][COUNT1].beta;
     IN[COUNTO][COUNT1].r=OUT[COUNTO][COUNT1].r;
     IN[COUNTO][COUNT1].s=OUT[COUNTO][COUNT1].s;
     IN[COUNTO][COUNT1].c=OUT[COUNTO][COUNT1].c;
     IN[COUNTO][COUNT1].z=OUT[COUNTO][COUNT1].z;
     IN[COUNTO][COUNT1].d=OUT[COUNTO][COUNT1].d;
     IN[COUNTO][COUNT1].delta=OUT[COUNTO][COUNT1].delta;
     IN[COUNTO][COUNT1].x=OUT[COUNTO][COUNT1].x;

}
}
```

```
/*Simulator Code Generated by Gensyssim*/

#include <stdio.h>
#define DIMO 5
#define DIM1 5
#define D_TIME_STEP 1

struct INOUTSTRUCT
{

        float x;
        float c;
        float s;
        float d;
        float delta;
        float z;
        float beta;
        float r;


}
IN[DIMO][DIM1],
OUT[DIMO][DIM1];

int PETYPE[DIMO][DIM1];

int COUNTO;
int COUNT1;
int DCOUNT;

FILE *USERINPUT,*USEROUTPUT, *fopen();

#include <pefile.h>

PAUSE()
{
        char c;
        printf("\nStrike N to Exit Any Other Key to Continue\n");
        while(!kbhit());
        c = getch();
        if((c == 'n')||(c == 'N'))
                exit(1);
}

main()
{
        INIT();
        while(1)
        {
        for(DCOUNT=0;DCOUNT<D_TIME_STEP;DCOUNT++)
        {
                INPUT();
                for(COUNTO=1;COUNTO<DIMO-1;COUNTO++)
                for(COUNT1=1;COUNT1<DIM1-1;COUNT1++)
```

```
                        switch(PETYPE[COUNTO][COUNT1])
                        {
                                case 0:{pe0();break;}
                                case 1:{pe1();break;}
                                case 2:{pe2();break;}
                                case 3:{pe3();break;}
                                default:{printf("pe undefined at PETYPE");
printf("[%d][%d]",COUNTO,COUNT1);}
                        }

                OUTPUT();
                EXCHANGE();
        }
        DISPLAY();
        PAUSE();
        }
}
```

Appendix E
Sample Simulator Execution

/*This file contains the output results of the square-root free Givens rotation operating on the input data matrix:

```
10    22    9
1     18    5
```

The input at the start of each cycle of computation is shown. Each variables output state is then shown at the end of a cycle of computation, beginning with the end of the first cycle of computation. A rectilinear array is seen but the active elements are arranged in a triangular array mapped to the rectilinear array. The array format is:

```
B    I    I
U    B    I
U    U    F
```

where B is a boundary cell, I is an internal cell, F is the final cell, and U are unused cells.

The input matrix is entered into the array in a skewed fashion:

```
T1      10    0    0
T2      1     22   0
T3      0     18   9
T4      0     0    5
```

where TX is a time step representing cycle of computation number X.*/

```
/*FIRST CYCLE OF COMPUTATION*/
USER INPUT
10.000000 0.000000 0.000000


SIN contents
0.100000  0.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


COSINE contents
0.000000  0.000000  0.000000
0.000000  1.000000  0.000000
0.000000  0.000000  0.000000


Z contents
10.000000  0.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
```

```
X contents
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


R contents
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000

RESIDUAL: 0.000000

/*SECOND CYCLE OF COMPUTATION*/
USER INPUT
1.000000 22.000000 0.000000


SIN contents
1.000000  0.100000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


COSINE contents
0.000000  0.000000  0.000000
0.000000  1.000000  1.000000
0.000000  0.000000  0.000000


Z contents
1.000000  10.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


X contents
0.000000  22.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


R contents
0.000000  2.200000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000

RESIDUAL: 0.000000

/*THIRD CYCLE OF COMPUTATION*/
USER INPUT
0.000000 18.000000 5.000000
```

```
SIN contents
0.000000  1.000000  0.100000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


COSINE contents
1.000000  0.000000  0.000000
0.000000  1.000000  1.000000
0.000000  0.000000  0.000000


Z contents
0.000000  1.000000  10.000000
0.000000  22.000000  0.000000
0.000000  0.000000  0.000000


X contents
0.000000  15.800000  5.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


R contents
0.000000  18.000000  0.500000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000

RESIDUAL: 0.000000

/*FOURTH CYCLE OF COMPUTATION*/
USER INPUT
0.000000 0.000000 9.000000


SIN contents
0.000000  0.000000  1.000000
0.000000  0.063291  0.000000
0.000000  0.000000  0.000000


COSINE contents
1.000000  1.000000  0.000000
0.000000  0.000000  1.000000
0.000000  0.000000  0.000000


Z contents
0.000000  0.000000  1.000000
0.000000  15.800000  22.000000
0.000000  0.000000  0.000000


X contents
```

```
0.000000  0.000000  8.500000
0.000000  0.000000  5.000000
0.000000  0.000000  0.000000


R contents
0.000000  18.000000  9.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000

RESIDUAL: 0.000000

/*FIFTH CYCLE OF COMPUTATION*/
USER INPUT
0.000000 0.000000 0.000000


SIN contents
0.000000  0.000000  0.000000
0.000000  0.000000  0.063291
0.000000  0.000000  0.000000


COSINE contents
1.000000  1.000000  1.000000
0.000000  1.000000  0.000000
0.000000  0.000000  0.000000


Z contents
0.000000  0.000000  0.000000
0.000000  0.000000  15.800000
0.000000  0.000000  0.000000


X contents
0.000000  0.000000  0.000000
0.000000  0.000000  8.500000
0.000000  0.000000  0.000000


R contents
0.000000  18.000000  9.000000
0.000000  0.000000  0.537975
0.000000  0.000000  0.000000

RESIDUAL: 0.000000
/*SIXTH CYCLE OF COMPUTATION*/
USER INPUT
0.000000 0.000000 0.000000


SIN contents
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
```

```
0.000000  0.000000  0.000000


COSINE contents
1.000000  1.000000  1.000000
0.000000  1.000000  1.000000
0.000000  0.000000  0.000000


Z contents
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000


X contents
0.000000  0.000000  0.000000
0.000000  0.000000  0.000000
0.000000  0.000000  8.500000


R contents
0.000000  18.000000  9.000000
0.000000  0.000000  0.537975
0.000000  0.000000  0.000000

RESIDUAL: 8.500000
/*END OF SAMPLE RUN*/
```

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, solid state sciences, electromagnetics and electronic reliability, maintainability and compatibility.

Printed by
United States Air Force
Hanscom AFB, Mass. 01731